НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ

«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет інформатики та обчислювальної техніки

Кафедра обчислювальної техніки

«На правах рукопису»

УДК _____

«До захисту допущено»

Завідувач кафедри

_____ *Стіренко С.Г.*

(підпис) (ініціали, прізвище)

“_____” _____ 20***20*** р.

Магістерська дисертація

зі спеціальності: *121. Інженерія програмного забезпечення*

(код та назва напряму підготовки або спеціальності)

Спеціалізація: *121.Програмне забезпечення високопродуктивних комп'ютерних систем та мереж*

на тему: *Метод підвищення ефективності навчання з підкріпленням для роботизованого агента*

Виконав: студент 6 курсу, групи ІП-83МН

(шифр групи)

Фінчук Олександр Васильович

(прізвище, ім'я, по батькові) (підпис)

Науковий керівник проф., д.т.н., проф. Стіренко С. Г.

(посада, науковий ступінь, вчене звання, прізвище та ініціали) (підпис)

Консультант _____

(назва розділу) (посада, вчене звання, науковий ступінь, прізвище, ініціали) (підпис)

Рецензент _____

(посада, науковий ступінь, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій магістерській дисертації немає запозичень з праць інших авторів без відповідних посилань.

Студент _____

(підпис)

Київ – 2020 року

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

Факультет (інститут) *Інформатики та обчислювальної техніки*
(повна назва)

Кафедра *Обчислювальної техніки*
(повна назва)

Освітньо-кваліфікаційний ступінь *магістр*
(назва ОКР)

Спеціальність *121. Інженерія програмного забезпечення*
(код і назва)

Спеціалізація *121. Програмне забезпечення високопродуктивних комп'ютерних систем та мереж*
(код і назва)

ЗАТВЕРДЖУЮ
Завідувач кафедри
Стіренко С.Г.
(підпис) (ініціали, прізвище)
« » _____ 20***20*** р.

ЗАВДАННЯ
на магістерську дисертацію студенту
Фінчуку Олександру Васильовичу
(прізвище, ім'я, по батькові)

1. Тема дисертації *Метод підвищення ефективності навчання з підкріпленням для роботизованого агента*

Науковий керівник дисертації Стіренко С. Г. д.т.н., проф.
(прізвище, ім'я, по батькові, науковий ступінь, вчене звання)

затверджені наказом по університету від «***08***» 03 ***2020*** р.№ ***3770-с***

2. Строк подання студентом дисертації _____

3. Об'єкт дослідження процес машинного навчання з підкріпленням

4. Предмет дослідження методи навчання нейронних мереж для навчання підкріпленням, віртуальне середовище для роботизованого агента

5. Перелік завдань, які потрібно розробити:

1. Проаналізувати сучасні методи машинного навчання з підкріпленням та існуючі середовища для тестування.

2. Розробити метод, який покращить швидкість та якість навчання

3. Реалізувати розроблений метод та створити віртуальне середовище для роботизованого агента.

4. Провести порівняльний аналіз розробленого методу із існуючими

6. Консультанти розділів дисертації:

Розділ	Прізвище, ініціали та посада консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання 03 лютого 2020

Календарний план

№ з/п	Назва етапів виконання магістерської дисертації	Строк виконання етапів дисертації	Примітка
1	Аналіз предметної області, огляд літератури	12.02.2020	
2	Дослідження методів машинного навчання з підкріпленням	16.02.2020	
3	Дослідження віртуальних середовищ для дослідження	20.02.2020	
4	Проектування методу, що дозволив би підвищити швидкість та якість навчання	28.02.2020	
5	Опис особливостей реалізації методу	08.03.2020	
6	Розробка програмного прототипу	10.04.2020	
7	Тестування та аналіз роботи системи	20.04.2020	
8	Оформлення матеріалів дисертації	05.05.2020	

Студент _____ О. В. Фінчук
(підпис) (ініціали, прізвище)

Науковий керівник дисертації _____ С. Г. Стіренко
(підпис) (ініціали, прізвище)

РЕФЕРАТ

Структура і обсяг роботи: магістерська дисертація викладена на 87 сторінках, складається зі вступу, 4 розділів та висновку. Містить 27 рисунків, 1 таблицю, 7 формул, список використаних джерел із 10 найменувань.

Актуальність роботи.

Автоматизація процесів досягла досить високого рівня, проте залишилися задачі, які не можна описати алгоритмічно, тому, що рішення для них не знайдено, або ж занадто складне для опису. Для таких типів задач краще підходить машинне навчання з підкріпленням, оскільки там агент самостійно знаходить оптимальну поведінку без конкретного алгоритму за рахунок навчання. Такі задачі лише вимагають опису бажаної цілі.

Проте на даний момент такі агенти вимагають мільйони повторів для навчання, щоб розв’язати якусь примітивну задачу, яку людина може виконати практично одразу. Різницею виступає досвід людини протягом мільйонів років еволюції, що дозволив побудувати ефективні способи навчання, в першу чергу за рахунок аналізу. Якщо середовище стає занадто складним для агента, то жоден із існуючих методів не здатен забезпечити високу успішність навчання. Тому в даній магістерській дисертації планується пошук методів, які б змогли підвищити ефективність навчання для роботизованих агентів. Дані дослідження внесуть позитивний вклад у сферу штучного інтелекту, тому є доволі актуальними.

Мета роботи: підвищення швидкості та якості машинного навчання з підкріпленням для роботизованого агента.

Завдання досліджень:

1. Проаналізувати сучасні методи машинного навчання з підкріпленням та існуючі середовища для тестування.
2. Розробити метод, який покращить швидкість та якість навчання
3. Реалізувати розроблений метод та створити віртуальне середовище для роботизованого агента.
4. Провести порівняльний аналіз розробленого методу із існуючими

Об'єкт дослідження: процес машинного навчання з підкріпленням.

Предмет дослідження: методи навчання нейронних мереж для навчання підкріпленням, ефективність навчання з підкріпленням, віртуальне середовище для роботизованого агента.

Методи дослідження: методи статистичного опрацювання даних, теорія штучних нейронних мереж, теорія навчання з підкріпленням.

Ключові слова: штучні нейронні мережі, машинне навчання з підкріпленням, самонавчання, віртуальне середовище, OpenAI, роботизований агент, неперервне управління.

Публікації:

Фінчук О.В. СПОСІБ ПІДВИЩЕННЯ ЕФЕКТИВНОСТІ НАВЧАННЯ З ПІДКРІПЛЕННЯМ РОБОТИЗОВАНОГО АГЕНТА ЗА РАХУНОК ДЕКОМПОЗИЦІЇ ЗАДАЧІ. *Інтеграція науки та практики як механізм ефективного розвитку*: матеріали II міжнарод. наук.-практ. конф. (м. Київ, 24-25 квітня. 2020 р.). Київ, 2020. С.185-191.

Фінчук О.В. WEB MINING НА ОСНОВІ ЛЯМБДА АРХІТЕКТУРИ Rs Global, World science: випуск №7(35) том 2(м. Варшава, 12 липня 2018 р.) Варшава, 2018. С.4-8.

ABSTRACT

Structure and scope of master's thesis: master's thesis is presented on 87 pages, consists of introduction, 4 sections and conclusion. It contains 27 figures, 1 table, 7 formulas and a bibliography of 10 titles.

Relevance of work.

Automatization of daily processes showed great results, but still some tasks can’t be described with algorithms. Reason for that could be complexity of describing each possible corner case or simply solution hasn’t already discovered. For these types of tasks reinforcement learning will be more suitable, because agent finds optimal solution by itself without any specific algorithm. It achieves it by interacting with an environment. We only need to describe desired goal, which agent need to achieve.

However nowadays such agents require millions of repeats of similar episodes to resolve simple task, which human can solve almost immediately. Explanation for this could be that human mind were created by millions of years of evolution. That allowed to find efficient ways of learning, first of all analysis. If an environment become too complicated for an agent, then none of existing methods can’t resolve this quite efficiently. That’s why in this master’s thesis are planned to find a more efficient way of reinforcement learning for robotic agents. The research should make a positive contribution in artificial intelligence domain so it is quite relevant.

Purpose of work: increase velocity and success rate of reinforcement learning for a robotic agent.

Research tasks:

1. Analyze modern methods of reinforcement learning and existing virtual environments.

2. Develop a method to improve velocity and success rate of reinforcement learning.
3. Implement designed method and create virtual environment for a robotic agent.
4. Perform testing and analyze results. Compare results with existing solutions.

Object of research: process of reinforcement learning.

Subject of research: reinforcement learning methods, reinforcement learning performance, virtual environments for a robotic agent.

Research methods: methods of statistical data processing, theory of artificial neural networks, theory of reinforcement learning.

Keywords: artificial neural networks, reinforcement learning, self-learning, virtual environment, OpenAI, robotic agent, continuous control.

Publications:

Finchuk O. METHOD TO IMPROVE REINFORCEMENT LEARNING PERFORMANCE FOR A ROBOTIC AGENT USING DECOMPOSITION OF A TASK. Integration of science and practice as a mechanism of efficient development: materials of II international science and practice conference (Kyiv April 24-25, 2020). Kyiv, 2020. P.185-191.

Finchuk O. WEB MINING BASED ON LAMBDA ARCHITECTURE Rs Global, World science: publishing №7(35) vol. 2(Warsaw, July 12 2018) Warsaw, 2018. P.4-8.

ЗМІСТ

ВСТУП

Людство завжди прагнуло відкривати щось нове, створювати речі, небачені раніше. Від самого початку ми стараємося покращити наші життя оптимізувавши, автоматизувавши речі навколо. Епоха інформаційних технологій зробила великий прорив у автоматизації, бо дуже багато стандартних задач можна було описати за допомогою алгоритмів, проте останнім часом задачі, які хоче вирішити людина виявляються все складнішими і розв'язати їх за допомогою алгоритмів стає дуже важко, або навіть не можливо при тому, що людина здатна сама розв'язати їх. Саме тому цікавість до самопізнання зросла ще більше. Вчені частково дослідили як працює мозок людини і захотіли відтворити цей процес і відтоді почалися дослідження в області машинного навчання.

На даний момент уже зроблений деякий прорив у машинному навчанні з учителем, але найбільший інтерес становить навчання з підкріпленням, оскільки саме тут агент сам вивчає світ самостійно. Такі агенти уже здатні виконувати примітивні задачі у віртуальному середовищі, проте для цього їм необхідно навчатися протягом сотень мільйонів ітерацій, в той час як людина здатна просто поглянути на умови задачі, щоб почати її робити доволі успішно, а все через те, що за мільйони років еволюції людський інтелект став доволі розвинений і може проводити аналіз базуючись на уже існуючих категоріях.

Метою даної магістерської дисертації дослідження процесу навчання з підкріпленням роботизованого агента у віртуальному середовищі а також пошук методів оптимізації цього процесу, як приклад, зменшення кількості ітерацій для досягнення тієї ж ефективності виконання задачі

Для досягнення поставленої мети у проекті необхідно вирішити наступні завдання:

- Налаштувати віртуальне середовище для тренування агента
- Визначити задачу яку буде розв’язувати агент
- Розробити функцію винагороди та знайти методи її покращення
- Провести аналіз ефективності процесу навчання

Об’єктом дослідження є процес навчання роботизованого агента з підкріпленням

Предметом дослідження є методи навчання нейронних мереж для навчання підкріпленням, ефективність навчання з підкріпленням, віртуальне середовище для роботизованого агента.

ЗАГАЛЬНІ ПОЛОЖЕННЯ

1.1. Опис предметного середовища

Автоматизація займає велике місце в людській діяльності. За останні роки дуже багато задач було розв’язано за допомогою описаних алгоритмів поведінки, проте деякі задачі можна розв’язувати без опису алгоритму дій. Галуз інформатики в якій поведінка отримується за рахунок статистичного навчання без явного опису алгоритму поведінки називається машинним навчанням. Суть такого підходу полягає у тому, що з часом програма покращує свою поведінку P у певній задачі W в залежності від отриманого досвіду E.

Виділяють такі основні категорії машинного навчання:

- Машинне навчання з учителем – категорія у якій навчання проводиться із додатковою інформацією зі сторони – учителем. В такому навчанні системі, що навчається уже дають вхідні дані та правильні результати відповідно до цих даних, а система статистичним способом повинна знайти найбільш близьку поведінку, до тієї, яка відображає зв’язок між вхідними та вихідними даними. Для прикладу це може бути передбачення запізнення авіарейсу, чи класифікація зображень. Перевагою даного методу є те, що дані уже підготовленні для навчання. Проте недоліком є те, що навчена система не може поводити себе краще, ніж їй було показано учителем.
- Машинне навчання без учителя – категорія машинного навчання у якій невідомо нічого про вхідні дані, тому дана категорія розв’язує зовсім іншу категорію задач. Це може бути пошук залежностей у самих даних завдяки кластеризації.
- Машинне навчання з підкріпленням – дана категорія найбільш відповідає тому, як навчаються живі істоти в реальному світі. Навчання

проводиться завдяки дослідженню простору і отримані винагороди, а метою такого навчання є пошук поведінки, яка приносила б найбільшу винагороду. Для прикладу це може бути автономний робот, що навчається ходити, або ж система, що вчиться грати в ігри. Такий тип навчання може бути поєднаний із навчанням з учителем, де на перших етапах системі надають якісь приклади поведінки. Саме з даним типом буде проводитися дослідження у цій магістерській дисертації.

Реалізація навчання може бути досягнена різними способами. Одним із способів є лінійна або нелінійна регресія, де на основі вхідних даних та їх мітки знаходиться функція залежності із певними коефіцієнтами для кожного із многочленів рівняння. Наприклад $p = ax_0 + bx_1 + c$, де x_1 та x_2 вхідні параметри, p це мітка, а саме навчання зводиться до пошуку коефіцієнтів a, b, c. Іншим варіантом є дерево рішень у якому рішення виступає як вершина графу, а наслідком є його дочірні вершини. Листи відображають фінальну ситуацію після прийняття усіх рішень. Проте найбільш цікавим методом є глибоке навчання, яке використовує у собі штучну нейронну мережу, що подібна до будови людського мозку. Основною перевагою цього методу є те, що він дозволяє знаходити складні нелінійні залежності між вхідними даними та мітками про них. Ідеєю створення цього методу було пошук методів рішення задач так, як це роблять істоти і саме тому архітектура була запозичена після відкриття особливих клітин мозку, які передавали сигнали одна одній при розпізнаванні звуку чи зображення.

Така система дозволяє вирішувати певні задачі статистичним шляхом, оскільки всередині штучної мережі розташовані нейрони із певними коефіцієнтами і активуються в залежності від них. Внутрішню систему можна змінювати і відповідно вона почне видавати інші результати. Розглянемо архітектуру штучної нейронної мережі:

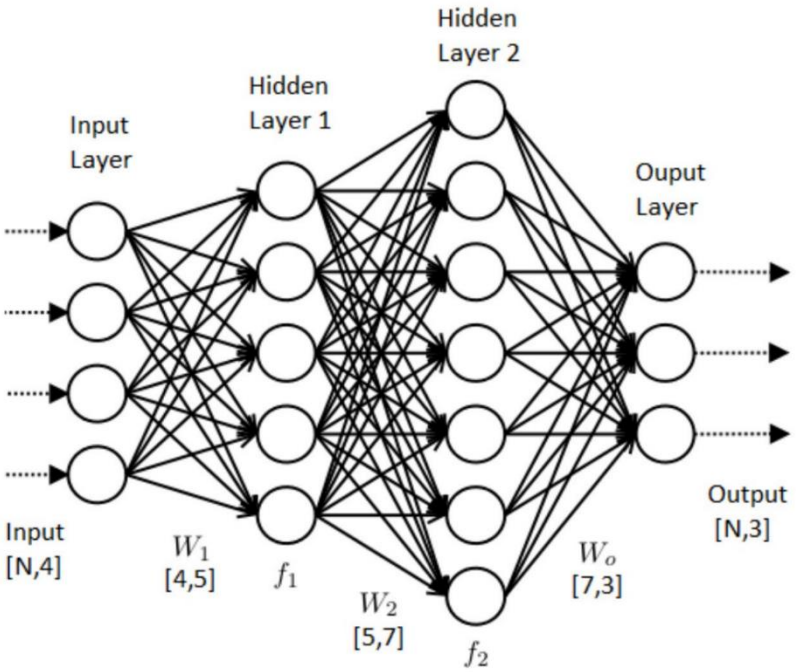

Рисунок 1.1 – схема штучної нейронної мережі.

Штучна нейронна мережа складається із нейронів, як виступають у якості вершин графа, що зв’язані між собою. Ці нейрони згруповані між собою у 3 основні шари:

- Шар вхідних параметрів отримує на вхід масив усіх параметрів на яких навчається система для одного випадку.
- Шар вихідних параметрів відображає результат роботи нейронної мережі за рахунок активації певних нейронів.
- Прихований шар відповідає за всю логіку прийняття рішень за рахунок зв’язків між нейронами.

Кожен нейрон має свою функцію активації *f*, яка змінює вхідний сигнал. Змінений сигнал передається кожному вихідному нейрону із врахування коефіцієнту, ваги ребра $w_{ijk.}$ В результаті у нейрон входить декілька ребер, усі вони агрегуються і обчислюється єдине значення для входу. Дане значення обчислюється завдяки функції поширення, що має вигляд:

$$p_j = \sum_{i=0}^{n} f_{out}(i) * w_{ij}$$

Отримане значення p_j відображає вхідне значення для нейрону j у певному рівні, а f_{out}(i) відображає функцію обчислення вихідного значення з попереднього нейрону.

Таким чином можна проходити усю нейронну мережу починаючи з вхідного шару і до шару вихідних параметрів, обчислюючи значення кожного з нейронів. Цей процес називається пряме поширення і дозволяє отримувати вихідні значення, що можуть відображати певну поведінку класифікацію, чи передбачення. Коректна робота із високим відсотком успішності може бути тільки за умови правильно розставлених коефіцієнтів та інших параметрів мережі. Коефіцієнти змінюються під час навчання.

Зворотне поширення помилки – це метод для зміни коефіцієнтів нейронної мережі на основі обрахунку передбаченого мережею значення та порівняння його з очікуваним. Після обчислення значення останнього шару вони порівнюються із мітками для вхідних даних і вираховується на скільки потрібно змінити ваги, щоб наблизити очікуваний результат, після цього можна відновити бажаний попередній шар і далі уже від нього обновлювати усі попередні коефіцієнти. Процес коригування ваг можна відобразити за допомогою формули:

$$w_{ij}(k+1) = w_{ij}(k) + \alpha * \frac{\Delta f_{loss}(X, w_{ij}(k))}{\Delta w_{ij}(k)}$$

Дана функція відображає градієнтний спуск для пошуку локального мінімуму при якій функція втрати буде видавати найменше значення. w_{ij} *(k+1)* означає вагу нову вагу на наступному кроці *k*. Коефіцієнт *a* слугує

для швидкості градієнтного спуску, потрібно його обирати оптимальним, щоб навчання було занадто повільним, але і щоб було достатньо точним, оскільки при занадто великому a можна пропустити локальний мінімум. $f_{loss}(X,w_{ij}(k))$ є функцією втрати і процес навчання зводиться до мінімізації цієї функції. На вхід функція отримує список вхідних параметрів для даного шару X та усі ваги ребер. ЇЇ можна виразити як:

$$f_{loss}\left(X, w_{ij}(k)\right) = \frac{1}{2n}\sum_{i=1}^{n}(y_{exp} - y_{res})^2$$

Значення y_{exp} очікувані значення і беруться із наданої множини даних для тренування, а y_{res} відображає отримані значення і вираховується на основі вхідних параметрів x та ваг w. Параметр n відповідає за кількість нейронів у поточному шарі.

1.2. Огляд наявних аналогів

Основною задачею даної магістерської дисертації є навчання роботизованого агента, тому найбільшу цікавість становлять уже існуючі алгоритми навчання з підтримкою та віртуальні середовища на яких проводиться навчання та огляд задач які ставляться перед агентом. Навчання з підкріпленням може бути реалізовано різними способами, які відносяться до машинного навчання, проте більшу цікавість мають алгоритми, які в основі мають штучні нейронні мережі, оскільки таке навчання може давати кращі результати і подібне до того, як працює мозок живих істот. Жоден із існуючих алгоритмів не є універсальним і можу вирішувати усі типи задач машинного навчання з підкріпленням ефективно. Усі вони призначені для розв’язання певної групи задача. Наприклад у певних алгоритмах агенту може бути надана вся необхідна інформація про середовище і він використовуючи її повинен знаходити оптимальне рішення. У інших категоріях агент повинен самостійно усе

досліджувати і знаходити оптимальну поведінку на своїх дослідженнях. Також можна по різному оцінювати роботу агента і для різних алгоритмів тип такої оцінки може мати ключову роль. Основні алгоритми можна згрупувати за категоріями.

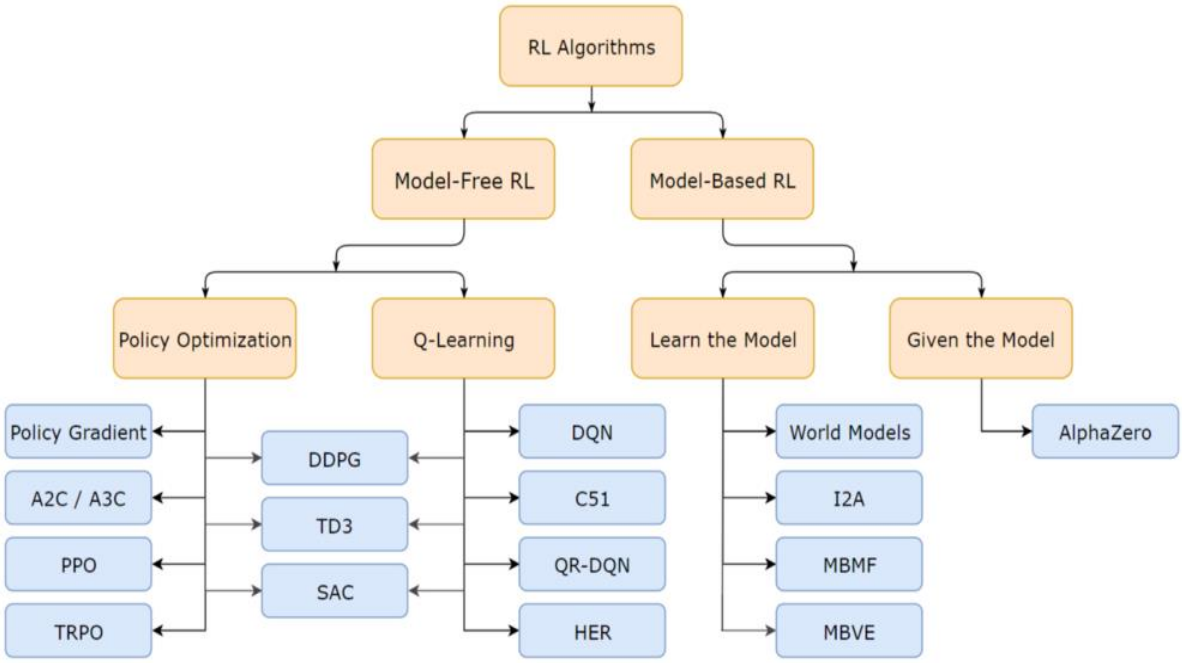

Рисунок 1.2 – основні алгоритми навчання з підкріпленням

На рисунку 1.3 зображено типи алгоритмів машинного навчання з підкріпленням та основні алгоритми даних типів. В першу чергу розглянемо різницю між алгоритмами з моделлю та без.

Алгоритми основані на використанні внутрішньої моделі використовують навчання для побудови моделі переходів і віддачі середовища. Як тільки модель побудована можна уже будувати рішення та планування базуючись на даній моделі.

З іншої сторони алгоритми без внутрішньої моделі націлені на пошук оптимального рішення маючи тільки політику поведінки або список нагород в залежності від стану в якому знаходиться агент. Такі алгоритми зазвичай є менш точними, оскільки інформація від середовища перемішана із попередньою яка може бути отримана внаслідок помилки агента. Але з

іншої сторони такі алгоритми більш універсальні. В свою чергу вони поділяються за ціллю навчання. Одні націлені оптимізацію поведінки за рахунок постійної оцінки дій, інші ж націлені на оптимізацію станів. Розглянемо деякі з них.

1.2.1. Алгоритми оптимізації політики поведінки

Дані алгоритми дозволяють агентам вчитися напряму базуючись на функції поведінки яка відображає стан у поведінку. Поведінка визначається без використання функції цінності. Існує два типи даних алгоритмів: з детерміновою поведінкою, та стохастичною поведінкою. В першому випадко відображення стану на дію відбувається без невизначеності, в іншому ж відображення використовує розподіл імовірностей для кожної із дій.

Policy Gradient

У цьому методі ми має поведінку π та параметр θ, В результаті π видає розподіл можливостей для кожної з можливих дій в даному стані.

$$\pi_\theta(a|s) = P[a|s]$$

Тоді ми повинні знайти найкращі параметри θ щоб оптимізувати функцію нагороди J(θ), маючи фактор знижки γ та нагороду r.

$$J(\theta) = E_{\pi\theta}[\sum \gamma r]$$

Алгоритм використовує градієнтний спуск для пошуку найкращих параметрів, які покращують поведінку

Asynchronous Advantage Actor-Critic (A3C)

Даний метод був розроблений Google’s DeepMind. Основні концепції:

Асинхронний: декілька агентів навчаються асинхронно у своїй власній копії середовища, а головний агент потім об’єднує зібраний досвід, таким чином досвід виходить більш варіативним, а процес навчання можна розпаралелити оскільки навчання відбувається для кожного агента незалежно.

Вигода: схоже на PG де функція оновлення використовує винагороди від набору досвіду щоб сказати агенту, які дії принесли користь, а які ні

Суб’єктно критичний: поєднує перевагу оптимізації політики поведінки та оцінки стану, оскільки використовує обидві функції цінності V(s) стану та поведінки π(s)

Trust Region Policy Optimization (TRPO)

Сенс методу полягає у тому, щоб робити дії, які максимально змінюють політику поведінки. Проте вводиться певна константа яка накладає обмеження на максимальне розходження між старою поведінкою та новою.

Може використовуватись у середовищах із дискретним простором дій або безперервним.

Proximal Policy Optimization

За основу взятий метод TRPO тому ідея схожа і також може використовуватись у із дискретним та безперервним простором дій.

Основна відмінність заключається у тому, що цей метод старається розв’язати проблему швидшого покращення політики поведінки без обвали продуктивності. В основі лежить метод контрольованого оновлення поведінки із мінімальними затратами на підбір параметрів навчання моделі і завдяки цьому цей метод став популярний, особливо після того як зробив

прорив у світі глибокого навчання з підкріпленням перемігши одних з найкращих гравців світу у Dota2

1.2.2. Q-learning або алгоритми базовані на оцінці стану

Метод Q-learning заключається у навчанні за допомогою функції оцінки стану Q(s, a). Ця функція дає оцінку наскільки була хороша дія прийнята в даному стані в загальному алгоритм виглядає так:

1. Ініціалізувати таблицю станів Q
2. Повторити доки не буде достатній рівень успіху
    2.1. Обрати дію базуючись на даному стані
    2.2. Виконати обрану дію
    2.3. Оцінити нагороду
    2.4. Оновити таблицю Q

Недоліком даного методу є те, що станів може бути занадто багато, особливо у безперервному середовищі.

Deep Q Neural Network (DQN)

Основною ідеєю є те, що таблиця станів замінюється глибокою нейронною мережею. Це дозволяє вирішити проблему, коли можливих станів занадто багато і таким чином спрощуючи модель економлячи час на навчанні

Hybrid

Основною задачею алгоритмів даного класу є поєднання переваг алгоритмів основаних на оптимізації політики поведінки та алгоритмів оптимізації оцінки стану, основними представниками є

- Deep Deterministic Policy Gradients (DDPG
- Soft Actor -Critic (SAC)
- Twin Delayed Deep Deterministic Policy Gradients (TD3)

1.2.3. Існуючі віртуальні середовища для навчання з підкріпленням

На даний момент декілька організацій розробило свої віртуальні середовища для навчання агентів. Спочатку це були ігри Atari, оскільки вони чудово підходили для пошуку оптимально поведінки у простому середовищі із простими стратегіями. Такі середовища теж дозволяли на інтуїтивному рівні розуміти чи агент поводиться як бажано.

OpenAI Gym

Організація OpenAI старається стандартизувати усі розробки в області навчання з підкріпленням, тому розробила єдиний інтерфейс для тестування алгоритмів навчання з підкріпленням та різноманітні середовища. Ці розробки активно використовуються дослідниками з усього світу. Це є позитивним фактором, оскільки дозволяє порівняти розроблений метод із існуючими у рівних умовах. Ось деякі з них

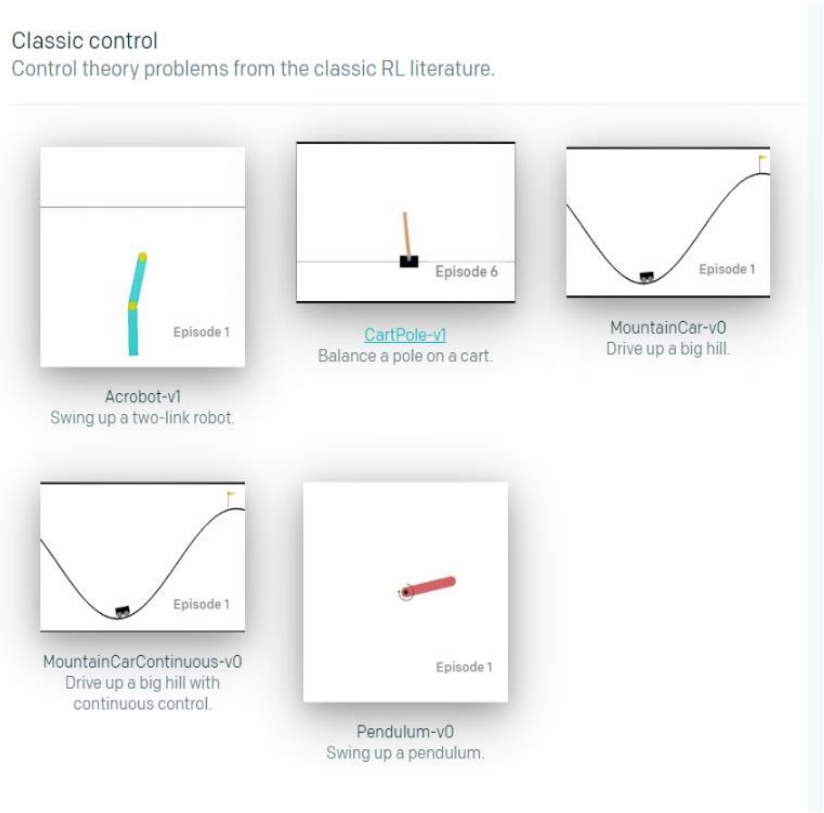

Рисунок 1.3 – середовища класичного контролю

В даних середовищах реалізовані прості задачі управління, щоб максимально сфокусуватися на розумінні алгоритмів навчання. Чудово підходить для ознайомлення із теорією машинного навчання з підкріпленням. Наприклад тут є задача для виїзду машини під гору або втримування палиці у вертикальному положенні.

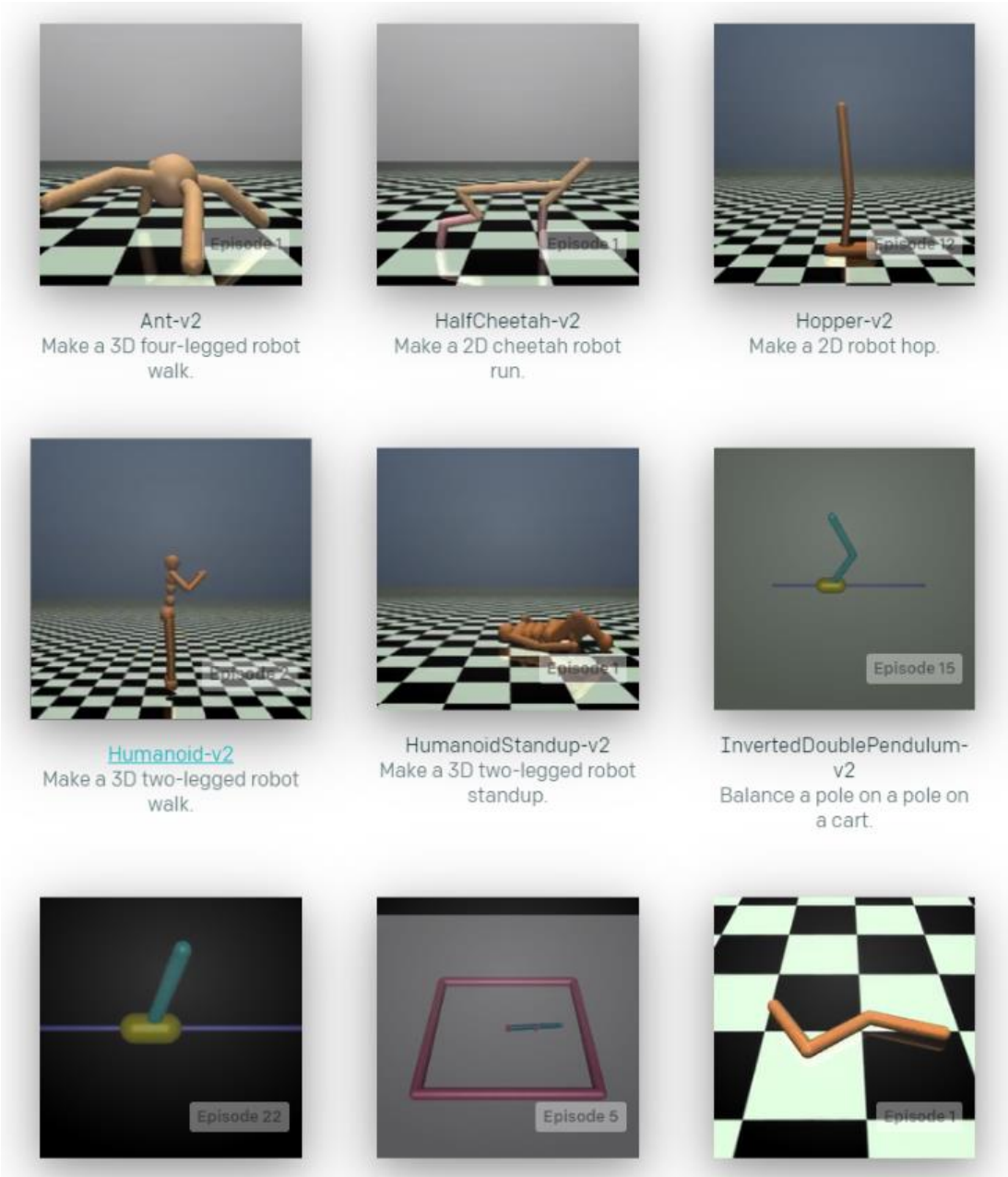

Рисунок 1.4 – MuJoCo средовища

В середовищах зображених на рисунку 1.5 використовується сторонній 3D світ із фізичними законами. Усі об’єкти можуть взаємодіяти

один з одним з поведінкою наближеної до реальної. Завдяки великому вибору можливих компонентів та зв'язків між ними можна створити велику множину можливих середовищ для тренувань. Наприклад можна натренована людиноподібного робота ходити, або робота схожого на пса бігати. Також дане середовище дозволяє генерувати велику кількість епізодів за секунду, що пришвидшить навчання. Недоліком є те, що моделювання відбувається на процесорі, а не на відеокарті, тому швидкість моделювання могла б бути вищою.

Unity ML Environments

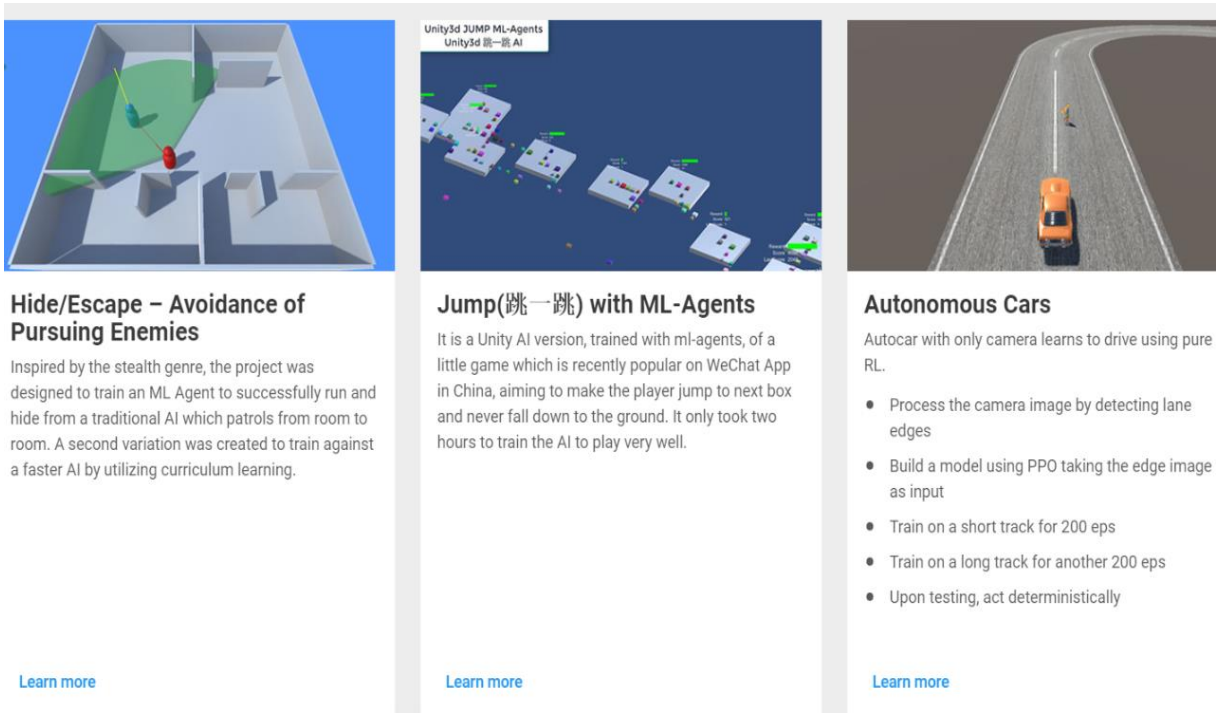

Рисунок 1.5 - Unity ML Environments

Середовище розробки ігор та віртуального 3D світу Unity давно завоювало популярність серед розробників і має доволі потужні інструменти для розробки віртуального середовища, тому розробників із світу машинного навчання це середовище теж зацікавило. Великою перевагою є те, що будь хто може поділитися свої середовищем тут. Найбільш популярними є гра в схованки, автономний автомобіль.

Отже, найбільш оптимальним віртуальним середовищем у якому можна тренувати роботизованого агента є Mujoco, оскільки задачі роботів зазвичай зводяться до взаємодії із реальним фізичними світом і об’єктами всередині нього. Можна використовувати напрацювання інших дослідників та розробників, щоб створити власне середовище.

1.3. Постановка задачі

1.3.1. Призначення розробки

Метою даної магістерської дисертації є пошук оптимальнішого методу навчання роботизованого агента в спеціальному середовищі. Це важливо для розвитку досліджень у галузі машинного навчання з підкріпленням, що є найбільш наближеним до того як навчаються істоти у реальному світі.

Після аналізу існуючих алгоритмів було виявлено, що усі вони вимагають величезної кількості ітерацій для навчання і успішність навчання дуже залежить від множини можливих станів та множини можливих дій. Чим ці множини більші, тим агенту важче знаходити оптимальну поведінку і тому успішність погіршується. В ході дослідження планується знайти методи для підвищення навчання ефективності агента навіть у складному середовищі.

Дані дослідження можна бути використати у подальших дослідженнях які ще більш наблизять агентів до розуміння навколишнього світу. Також готове рішення може бути використане іншими студентами для пришвидшеного вивчення даної галузі.

1.3.2. Цілі та задачі розробки

В ході роботи планується налаштувати віртуальне середовище, яке буде відповідати наступним вимогам:

- Будуть застосовані прості фізичні закони

- Агент повинен взаємодіяти із середовищем через єдиний інтерфейс
- Агент може спостерігати середовище завдяки різноманітним датчикам включаючи камеру
- В середовищі повинен визначатися початковий стан
- Повинен визначатися кінцевий стан до якого буде прагнути агент

Після налаштування середовища планується дослідження навчання агента у цьому середовищі, перевірки його ефективності.

Далі буде проведена розробка ефективнішого методу навчання а також буде розроблена продуктивна функція оцінки користі прийнятої дії. Сам алгоритм має покращувати успішність навчання в порівнянні з існуючими. Також швидкість навчання повинна бути збільшена. Саме ці два фактори і визначають ефективність. Ще однією вимогою є пошук методу, щоб згадані покращення працювали на більш складних середовищах і масштабувалися.

ВИСНОВОК ДО РОЗДІЛУ 1

В даному розділі було розглянено усі необхідні теоретичні відомості для подальшого дослідження. Також було проведено аналіз існуючих алгоритмів для навчання з підтримкою. На даний момент переважно усі розробки в сфері машинного навчання з підкріпленням ведуться з використанням певних віртуальних середовищ, саму тому було розглянуто основні із існуючих.

В ході аналізу існуючих рішень було вирішено, що переважна більшість алгоритмів намагається покращити ефективність навчання за рахунок різних методик, проте усі вони вимагають мільйони ітерацій для навчання і ефективність дуже падає при великій множині можливих станів середовища та великому вибору можливих одночасних дій агента. Тому було поставлено вимоги до методу, що розробляється.

Розроблений метод навчання з підкріпленням повинен відповідати наступним вимогам:

- показувати кращу успішність виконання поставленої задачі
- саме навчання повинно займати меншу кількість ітерацій і відповідно менше часу
- розроблений метод повинен масштабуватися на складніші середовища

Вимоги до віртуального середовища:

- Середовище повинне дозволяти моделювати різні задачі для дослідження
- Середовище повинне масштабуватися на складніші задачі

ПРОЕКТУВАННЯ СИСТЕМИ

2.1. Проектування віртуального середовища

Оскільки навчання агента потребує мільйони ітерацій навчання, то навчання в режимі реального часу буде тривати доволі довго і для цього необхідно мати реального робота. Перевагою віртуального середовища є те, що створення цього середовища не потребує додаткових фізичних ресурсів, модель середовища повністю визначена і формалізована, оскільки є спрощеною і фокусується тільки на важливих елементах. Також навчання на ньому проходить швидше, оскільки час можна пришвидшити, а саме середовище можна скопіювати і отримати декілька паралельних середовищ, результати навчання яких можна об’єднувати. Також при такому підході умови середовища можна доволі швидко змінювати або навіть саму модель.

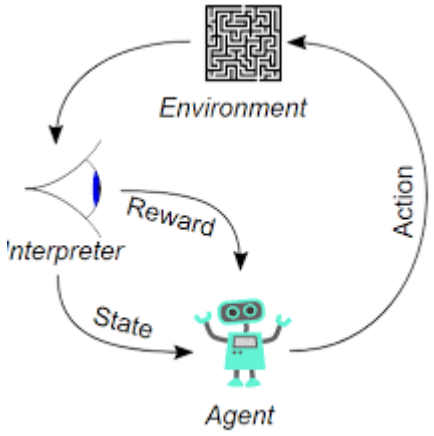

Рисунок 2.1 – взаємодія агента із середовищем

На рисунку 2.1 зображені основні компоненти взаємодії агента та середовища. Взаємодія повинна проходити за допомогою однозначно визначеного інтерфейсу, щоб у разі зміни середовища спосіб комунікації не змінився. В першу чергу агент повинен щось спостерігати, стан в якому він знаходиться. Це може бути як зображення з віртуальної камери,

сигнали з віртуальних датчиків так і більш специфічна інформація як положення у трьох вимірному просторі.

Базуючись на спостережені агент повинен могти оцінити наскільки даний стан середовища хороший і яку винагороду може принести. Наприклад агент просунувся на 10% ближче до цілі, що є хорошим результатом, проте він перевернув предмет, який не повинен був чіпати, що є поганим результатом.

Також у агента завжди є якась ціль до якої він повинен старатися дійти. Як тільки агент доходить до цієї цілі моделювання повинне бути обновлене. Наприклад агент дібрався до визначеної точки у просторі за найменший час.

Щоб обновити моделювання у моделі має бути визначений початковий стан, до якого буде зроблене повернення при досягнені агентом цілі або при його невдачі. Для прикладу агент буде повернений на стартову пряму, а усі елементи з якими він взаємодіяв на свої початкові позиції.

Середовище повинне змінюватись в залежності від часу та дій, які виконує агент, тобто якщо агент дає команду на рух вперед, то якщо перед ним є рухомий предмет, то агент змінить і своє положення і положення предмету.

Окрім стану середовища, агент також повинен отримувати список можливих дій з яких він може обирати. Якщо справа від агента розташована нерухома стіна, то агент може рухатись тільки вліво.

Не менш важливо відображати поточний стан середовища для людини у зручному для неї вигляді, щоб у будь яким момент можна було спостерігати як відбувається процес навчання або наскільки добре справляється із задачею уже натренований агент. Щоб була можливість

поглянути на процес навчання потім, середовище повинне мати можливість запису процесу навчання у зжатому вигляді на диск для подальшого відтворення.

В першому розділі було проаналізовано, що доцільніше буде використовувати OpenAI gym для налаштування поставлених вимог. Дана розробка має зручний розроблений інтерфейс для роботи із різноманітними алгоритмами навчання та дозволяє під'єднувати нові середовища за допомогою інтерфейсу. У магістерській дисертації планується досліджувати поведінку робота, тому необхідна симуляція фізичного світу. У першому розділі було виявлено, що найкращим рішенням буде MuJoCo. Опис початково стану описується завдяки XML. У описі вказується форма кожного об'єкту, його розміри, позиція у просторі, матеріал та текстури. Матеріал можу бути не тільки жорстким на 100%, а й пружним, що може ще більше ускладнити задачу для агента, який маніпулюватиме такими об'єктами. Також об'єкти можна пов'язувати між собою за допомогою фіксацій. Таке з'єднання може бути гнучким і управлятися зі сторони якимось алгоритмом. У нашому випадку це буде алгоритм поведінки агента на основі натренованої нейронної мережі. Для даного дослідження оберемо дві наступні задачі, які необхідно буде описати у віртуальному середовищі:

- Роботизована рука, з 5ма пальцями, кожен з яких має 3 точки згину і може обертатися відносно лодоні. Рука буде зафіксована у просторі і зможе рухатися тільки вліво, вправо, вниз, вверх відносно зафікованої осі. На поверхню руки буде розташовано об'єкт для обертання.

- Робот маніпулятор подібний до тих, що використовуються на виробництві. Його положення теж буде зафіксоване, проте управління буде здійснюватися завдяки 5 точкам згину та повороту.

2.2. Алгоритм навчання агента

У навчанні з підкріпленням досвід для оновлення нейронної мережі агент генерує самостійно, без учителя.

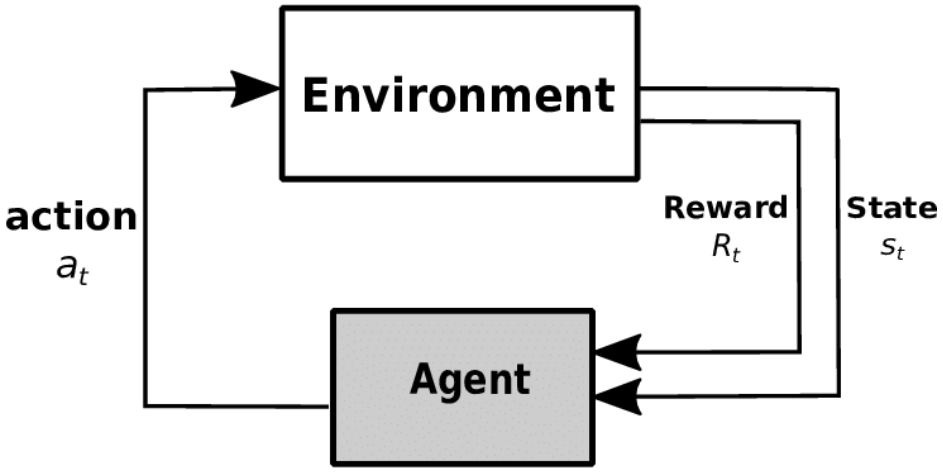

Рисунок 2.2 – схема роботи машинного навчання з підкріпленням

Під час навчання з підкріпленням основними компонентами є агент, та середовище з яким він взаємодіє. Кожну ітерацію агент виконує дію a_t відповідно до політики поведінки p_t , а середовище дає сигнали для агента. Такими сигналами може бути зміна стану середовища s_t та винагорода за зроблену дію r_t. Протягом часу агент взаємодіє із середовищем і покращує свою поведінку для збільшення отриманої винагороди за свої дії. На кожному етапі агент може зробити якусь дію із множини можливих дій A, а імовірність вибору кожної із дій визначається самим агентом і змінюється протягом навчання. Таким чином для кожної зміни стану в певний момент часу існує імовірність, яку можна виразити як $p(s_{t+1} \mid s_t, a_t)$. Після прийнятої дії агент отримує винагороду яку залежить від поточного стану і прийнятої дії, тому в загальному її можна описати як $r : S*A \to R$.

Кожна нова ітерації починається із якогось початкового стану s_0 і агент робить дії і старається оптимізувати свою поведінку на основі отриманого досвіду, цей процес можна описати як

$$Q(s_t, a_t) = E[R_t|s_t, a_t]$$

Позначимо пошук оптимальної поведінки як ***Q*,*** а оптимальні стани як ***s'*** та оптимальні дії як ***a'.*** Тоді весь процес навчання з підкріпленням можна описати за допомогою рівняння Белмана:

$$Q^*(s,a) = E_{s' p(\cdot|s,a)}[r(s,a) + y * max_{a' \in A} Q^*(s', a')]$$

У даному рівнянні використано коефіцієнт ***y***, який відповідає за коригування впливу нового досвіду на навчання.

Опис функції винагороди є найбільш важливим і складним моментом, оскільки вона повинна бути максимально збалансованою та оптимальною. Якщо функція буде видавати занадто високу винагороду за короткочасну дію, то агент не досягне кінцевої мети, оскільки він застряне на одній фазі, яку буде постійно виконувати, оскільки це завжди приноситиме максимальну винагороду. З іншої ж сторони, якщо агент не отримуватиме винагороду майже ніколи, то він просто не розумітиме що йому потрібно робити і теж не зможе досягнути кінцевої цілі. Функції винагороди можна поділити на дві категорії:

- Функція винагороди, яка оцінює агента за кожен його крок в залежності від багатьох факторів. Такий підхід має перевагу, оскільки агент одразу починає отримувати якусь винагороду і рухатися в правильному напрямку, проте цей підхід і має деякі недоліки. В першу чергу це складність розробки такої функції, оскільки вона має передбачати усі можливі стани і знати оцінку для них. Це найбільш негативно виявляється у задачах у яких ми самі не знаємо яка поведінка оптимальна, тому можемо загнати агента у неправильні рамки.
- Функція винагороди, яка оцінює лише кінцеву винагороду, тобто ми просто описуємо, що агент повинен досягнути без детальної

оцінки кожної його дії. Основна перевага у простоті опису цієї оцінки, проте агент може навчатися дуже довго і не знаходити правильного шляху. Ще один недолік такої поведінки, що вона може бути непередбачувана і в результаті не відповідати нашим вимогам. Наприклад ми описуємо задачу дібратися до певної точки через перешкоди. Агент після довгого навчання знайшов спосіб як у віртуальному середовищі пройти скрізь перешкоди через помилку у описі віртуального середовища.

Як бачимо описані функції поведінки є взаємовиключні і тому не можна сказати, що якась краща за іншу, кожна із них підходить для певного класу задач.

Проте визначення функції винагороди не єдиний важливий компонент у навчанні з підкріпленням. Те як саме буди проходити навчання, тобто алгоритм навчання грає не меншу роль. В першу чергу потрібно визначити політику дослідження нових можливостей, оскільки при політиці поведінки відповідно до свого досвіду агент помилково може застряти у якомусь локальному мінімумі в який його завів помилковий досвід. Для того, щоб цього не трапилося можна агенту давати можливість вибору випадкових дій, таким чином агент буде швидше досліджувати середовище. У цьому підході недолік у тому, що занадто випадкове дослідження може обвалити успішність навчання. Ще одним підходом є додавання шуму до спостереження, яке отримує агент, таким чином агент теж буде краще реагувати на зміну стану середовища.

Не менш важливим фактором є те, як агент трактує для себе отримані спостереження. Агент може отримувати покращення у поведінці як з негативного досвіду так і з позитивного. Також під час навчання повинна бути присутня певна стабілізація, щоб після певного часу успішність не змінювалася у великих межах. Також важливо щоб агент

покращував свою поведінку навіть, якщо він робив усі дії правильно, проте допустив єдину помилку і тому не досягнув бажаної мети

2.2.1. Алгоритм повторення досвіду з віртуальною ціллю

У середовищі, де існує дуже багато станів, навчання може тривати дуже довго через те, що агент може пробувати дуже багато варіантів дій, проте ні один із них не наближуватиме агента до цілі. Для розв'язання цієї проблеми був розроблений алгоритм HER(Hindsight Experience Replay) – ретроспеткивне повторення досвіду. Основна ідея заключається у віртуальній цілі, тобто після того, як агент зробить неправильну дію буде зроблене навчання з цілю, при якій зроблена дія була б правильною. Наприклад, робот має прийти в точку A[1;0] проте він прийшов у точку B[2;1] що є помилковим результатом, тому після того, як агент отримає негативну нагороду буде повторений процес навчання із позитивним досвідом, де ціллю уже буде точка B. Таким чином агент для себе зрозуміє як саме потрібно рухатись і це значно пришвидшить процес навчання. Процес повтору важливий тому, що якби віртуальна ціль завжди заміщувала реальну, агент ніколи б не отримував негативного досвіду, тому отримувавши ціль і робивши неправильні дії, агент вважав би, що все робить правильно.

Даний алгоритм базується на уже існуючих алгоритмах навчання без визначений правил поведінки. Наприклад DDPG, DQN, SDQN. Для повторення досвіду вибираються епізоди із уже отриманої симуляції. Ці епізоди можна обирати по-різному. Наприклад це можуть бути випадковий набір епізодів, або ж епізоди які ставалися тільки після поточного епізоду. Також цей алгоритм вимагає функцію винагороди, оскільки потрібно генерувати віртуальну ціль, то зручніше бути працювати із класом винагород, де дається фінальна оцінка, без проміжних етапів, оскільки

якщо давати віртуальну ціль для кожного кроку, то це може заплутати агента ще більше.

Опишемо роботу даного алгоритму :

- Налаштувати базовий алгоритм alg (DDPG або інший) з усіма необхідними для нього параметрами
- Створити сховище для зберігання епізодів для повтору rep
- Ввести кількість епізодів I та тривалість кожного епізоду T
- Для кожного епізоду i:
  - Визначити нову g та згенерувати початковий стан s_0
  - Для кожного моменту часу t:
    - В залежності від обраного alg визначити дію, яку прийме агент в залежності від стану та цілі $a_t = policy(s_t, g)$
    - Отримати спостереження від середовища виконавши дію a_t та перейти в новий стан s_{t+1}
  - Для кожного моменту часу t:
    - Обчислити винагороду за зроблену дію використавши функцію винагороди $rew_t = reward\_f(s_t, g, a_t)$
    - Зберегти дану зміну у сховище повтору rep
    - Визначити усі додаткові віртуальні цілі відповідно до стратегії S
    - Для кожної віртуальної цілі g_vir:
      - Визначити нову винагороду для цього переходу $rew\_vir_t = reward\_f(s_t, g\_vir, a_t)$
      - Зберегти дану зміну у сховище повтору rep
  - Розбити сховище повтору відповідно до розміру пакету на якому буде проходити один етап навчання

- Для кожної частини сховища повтору batch:
  - Виконати процес навчання відповідно до alg

2.2.2. Алгоритм навчання з декомпозицією задачі

В результаті роботи алгоритму з віртуальною ціллю можна спостерігати покращення ефективності навчання уже на перших епохах навчання. Проте при такому підході все ще існує проблема якщо задача доволі складна, то одразу напряму знайти рішення для неї може бути важко, це пов’язано з тим, що якщо агент робить дію, яка є частиною правильного рішення, проте до кінцевої цілі йому ще далеко, то винагороду він за цю дію отримає мінімальну. Отже, якщо розбити задачу на підзадачі і навчати агента на окремих простих задачах, то в результаті агент зможе справитися із кінцевою ціллю за меншу кількість ітерацій навчання.

Наприклад, роботу необхідно поставити кубик у точку А синьою стороною вперед. Отже цю задачу можна розбити на такі підзадачі:

- переміщення кубика в точку А
- Обертання кубика навколо окремих осей X, Y або Z

В даному дослідженні ставиться акцент на дослідження ефективності розбиття задачі і порівняння її з оригінальною поведінкою. Проте розбиття на підзадачі часто може бути не оптимальним і вимагає великого втручання зі сторони розробника. Тут постають ті ж проблеми, що і з функцією винагороди, яка оцінює кожен крок. Розбиття може бути складним для розробки.

Потенційно можливо розробити систему в якій агент буде проводити задачу кластеризації на своїх діях і спостереженнях від середовища, далі він сам буде виділяти спільні патерни поведінки, які використовувалися і відточувати їх окремо виносячи це в окрему нейронну

мережу. Також агент паралельно буде вчитися коли саме використовувати яку модель поведінки і таким чином він зможе знаходити найбільш оптимальне розбиття і активацію.

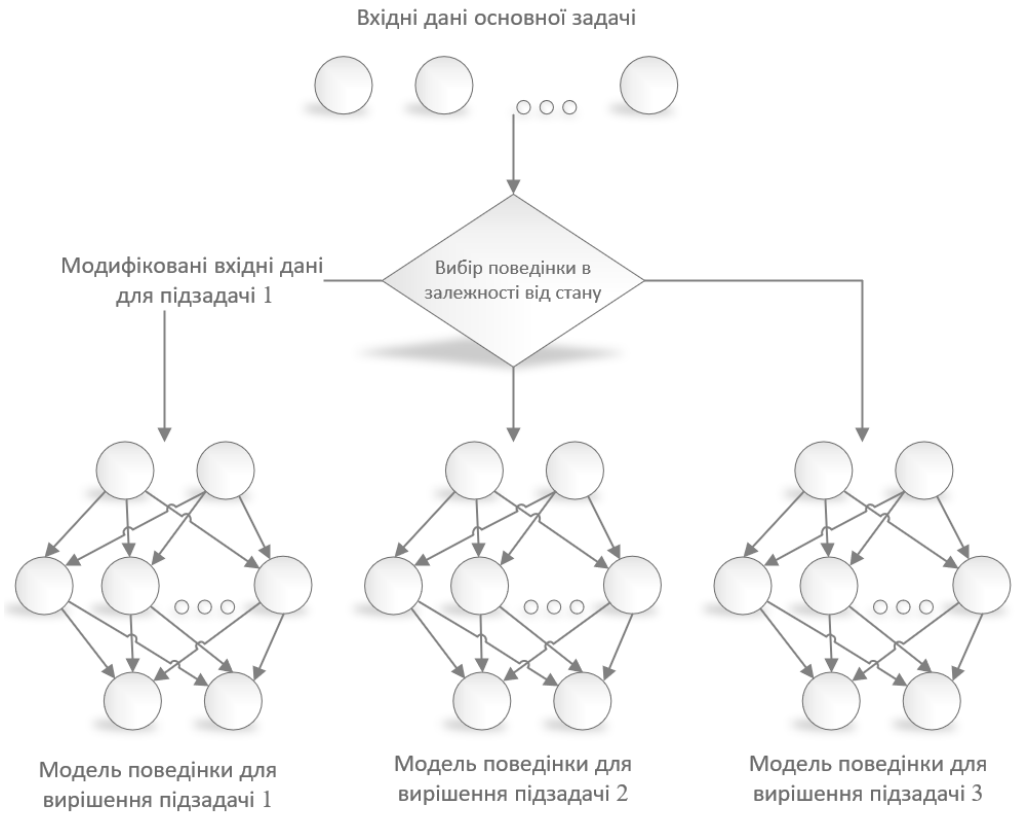

Рисунок 2.3 – схема декомпозиції задачі агентом

В запропонованому методі на рисунку 2.2 агент розбиває задачу якій планує навчитися і навчається кожній задачі окремо. Перевагою даного підходу є те, що агент може навчатися цим підзадачам паралельно в незалежності одна від одної. На вхід агент отримує поточний стан середовища та ціль яку необхідно досягнути. На основі цих даних агент обирає яку поведінку застосувати. Далі агент передає змінені вхідні параметри у дочірню модель поведінки і на її основі обирає яку дію прийняти. За рахунок спрощеної задачі агент буде швидше знаходити правильну поведінку. А завдяки поєднанню із алгоритмом повторення досвіду з віртуально ціллю ефективність буде ще вища. Таке поєднання

дозволить краще розв’язувати задачі, які вимагають неперервного контролю, оскільки множина можливих станів досить велика.

Опишемо алгоритм детально:

- Визначити базовий алгоритм для навчання підзадачам alg та створити всі необхідні компоненти для нього.
- Визначити функцію для декомпозиції задачі на менші підзадачі decomp_goal
- Визначити список підзадач sub_tasks, які необхідні для розв’язання поставленого класу задач main_task
- Для кожної підзадачі sub_task:
  - Якщо модель для розв’язання sub_task уже натренована, то виконати:
    - Завантажити модель поведінки у пам’ять і перейти до наступної підзадачі
  - Створити віртуальне середовище sub_env для розв’язання sub_task
  - Провести навчання для sub_task на базі створеного віртуального середовища sub_env з використанням базового алгоритму навчання alg
  - Зберегти натреновану модель у пам’ять і перейти до наступної підзадачі
- Створити середовище env для основної цілі main_task
- Для кожного епізоду E:
  - Обновити стан середовища до початкового s_0
  - Згенерувати нову ціль g, генерація повинна відповідати нормальному розподілу.
  - Розбити ціль на декілька цілей sub_goals відповідно до функції розбиття decomp_goal

- Визначити початкову підзадачу відповідно до першої цілі із sub_goals
- Для кожного для кожного моменту часу t:
  - Обрати модель поведінки $model_t$ відповідно до поточної підзадачі, яку агент розв'язує
  - На основі отриманих спостережень визначити дію, яку необхідно виконати $a_t = model_t(sub\_goal_t, s_t)$
  - Виконати a_t та отримати спостереження s_{t+1}
  - Обчислити винагороду за виконану дію $rew_t = reward(sub\_goal_t, s_{t+1})$
  - Якщо ціль підзадачі виконана перейти до наступної підзадачі і змінити поточну ціль на наступну

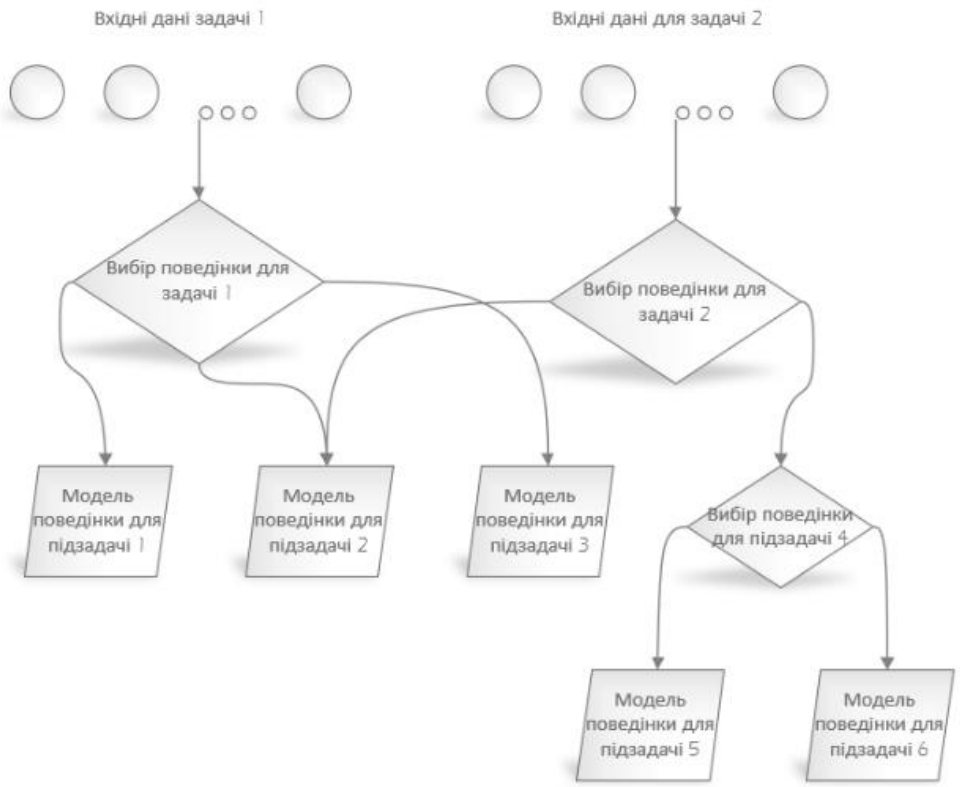

Рисунок 2.4 – комбіноване використання розробленого методу

В результаті розбиття на підзадачі агент вчиться якимось простим моделям поведінки. Завдяки запропонованому методу ці прості моделі можуть використовуватися для різних задач і комбінуватися у різних варіантах. Також вони можуть групуватися у різні рівні декомпозиції як зображено на рисунку 2.3. Наприклад агент навчився піднімати об’єкти та навчився ходити, а в іншій задачі він навчився водити авто і сам процес навчання був розділений на управління швидкістю та управління кермом. Таким чином після об’єднання усіх підзадач агент зможе виконувати доставку вантажу з використанням автомобіля, що є дуже складною задачею, проте декомпозиція задачі дозволила агенту навчитися усім цим задачам окремо, що зробило пошук рішення більш ефективним. Потенційною проблемою при такому підході може бути не співпадіння транзитного стану. Тобто кожна окрема підзадача очікує почати роботу у певному стані, проте попередня задача виконала ціль на 90% і тому початковий стан відрізнятиметься. Щоб вирішити це можна додавати певний шум до початкового стану та цілі, що точок перетину було більше.

2.3. Проектування середовища для тренування моделі

Як було зазначено у пункті 2.1 агент буде навчатися всередині віртуального середовища яке міститиме в собі елементи трьох вимірної графіки, тому вся система запускатиметься на машинах з дискретними відеокартами для підвищення швидкодії відображення віртуального середовища.

В середині моделі поведінки агента знаходитиметься нейронна мережа з великою кількістю прихованих шарів та вершин. Для тренування такої моделі необхідним буде глибоке навчання. Процес глибокого навчання вимагає потужних обчислень над матрицями, оскільки нейромережа містить велику кількість вершин, які необхідно оновлювати на кожному етапі навчання. Матричні операції найкраще виконуються на

відеокартах, оскільки вони були розроблені для відображення графіки, де потребується велика кількість множення матриць.

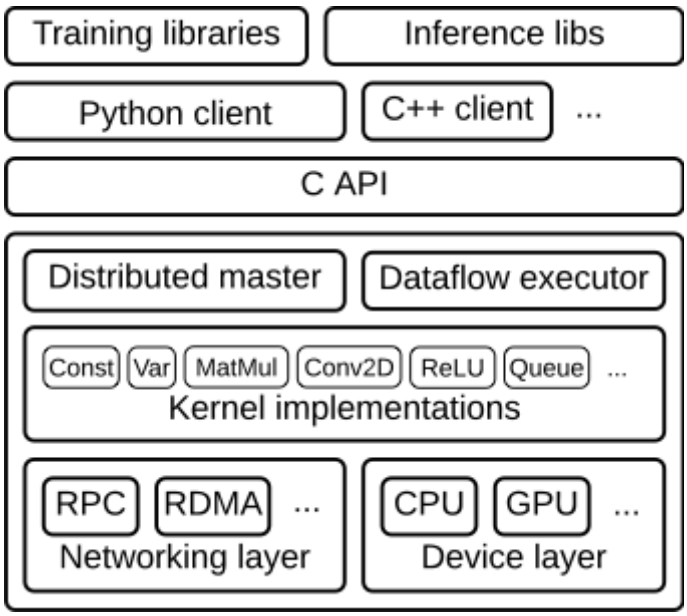

Рисунок 2.2 Компоненти TensorFlow

Для цієї цілі найкраще підходить бібліотека на Python – TensorFlow(Рис. 2.2) , оскільки має усі найнеобхідніші операції для роботи з машинним навчанням, такі як створення датасетів, створення тенсорів, використання найбільш поширених алгоритмів навчання і т.д.

Основними компонентами TensorFlow є

- Клієнт – визначає весь граф обчислення та ініціює його виконання
- Розподілений мастер
  - Виділяє підграф для обчислення
  - Розбиває підграф на частини для паралельного обчислення
  - Надсилає команди окремим робочим сервісам для обчислення
- Робочий сервіс

- виконує обчислення в залежності від апаратного забезпечення використовуючи реалізацію ядра
- Отримує та надсилає результати обчислень іншим робочим сервісам

- Реалізація ядра - містить в собі реалізацію базових алгоритмів та відповідає за виконання індивідуальних операцій

Для інтеграції TensorFlow з відеокартою буде використовуватись Nvidia Cuda – програмно-апаратна архітектура для забезпечення більш ефективних паралельних обчислень матриць завдяки реалізованих низькорівневих оптимізацій з використанням графічних процесорів компанії Nvidia. Оскільки в процесі роботи буде використовуватись глибоке навчання, то необхідний буде модуль cuDNN, що якраз за це відповідає.

ВИСНОВОК ДО РОЗДІЛУ 2

В даному розділі була спроектована модель тренування роботизованого агента у віртуальному середовищі. Проектування було розбито на наступні розділи

- Проектування віртуального середовища – були описані усі вимоги до віртуального середовища та запропоновані варіанти для задоволення цих вимог. Було вирішено використовувати комбінацію середовища для навчання нейронних мереж OpenAI Gym та середовища для моделювання фізичних процесів MuJoCo
- Проектування апаратно-програмного середовища для запуску віртуального середовища та тренування моделі. Було вирішено використовувати Nvidia Cuda та cuDNN для оптимізованого низькорівневого зв’язку з відеокартами Nvidia. Для навчання нейронних мереж буде використовуватись Tensorflow.
- Алгоритм оптимізації навчання агента – було запропоновано два способи оптимізації. Перший це спосіб підміни основної цілі віртуальною, щоб агент швидше навчався яка поведінка правильна, Другий це метод декомпозиції задачі на менші підзадачі, оскільки навчання простим задачам займає менше часу і їх можна розподілити.

РОЗРОБКА СИСТЕМИ

3.1. Розробка віртуального середовища

У другому розділі було обрано віртуальне середовище OpenAI Gym у поєднанні з віртуальним фізичним середовищем. У даному розділі буде описано основні компоненти для роботи з ним.

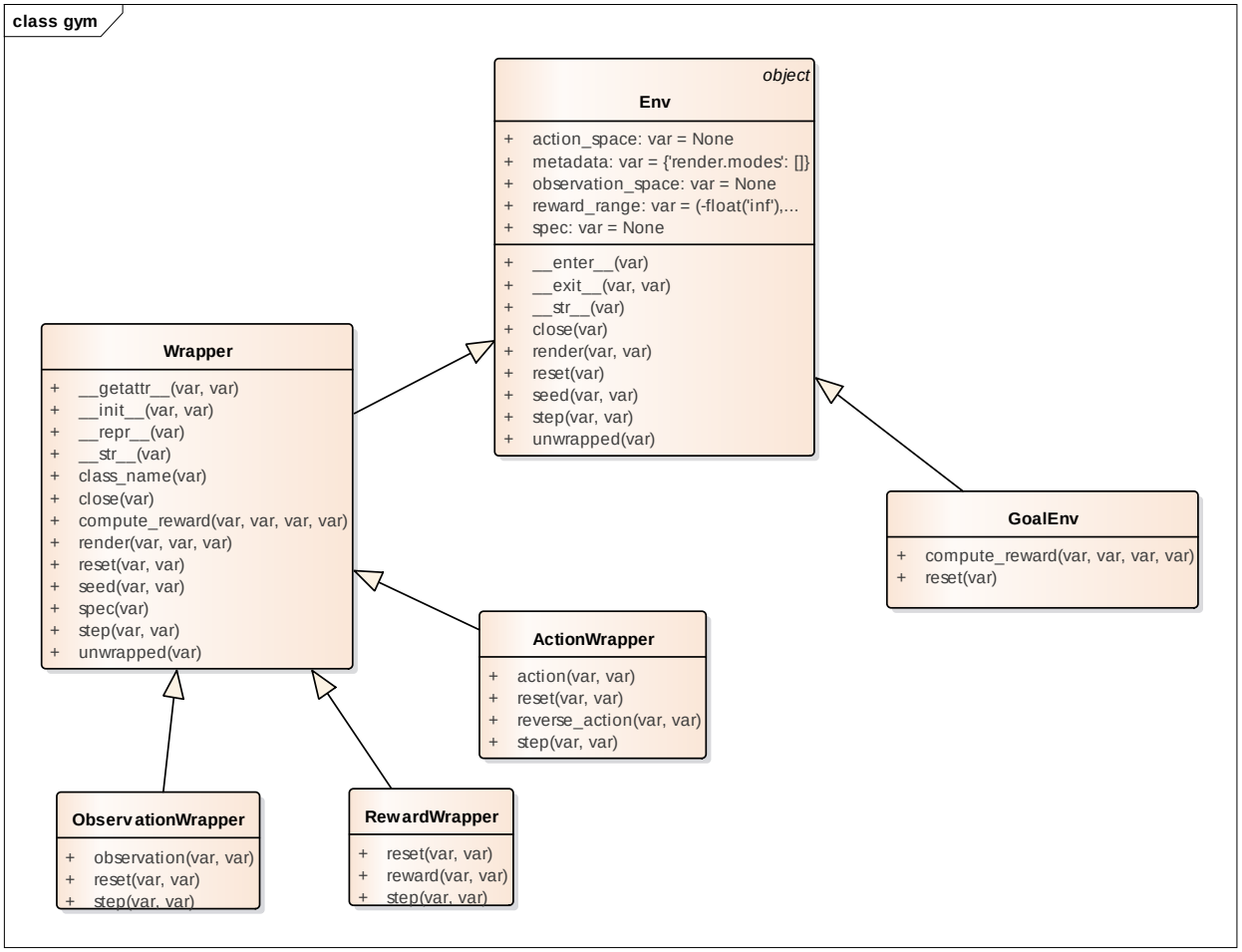

Рисунок 3.1 Діаграма класів для роботи з OpenAI Gym

Усі можливі середовища наслідуються від класу *Env*, який виступає основний інтерфейсом, або реалізують патерн декоратор маючи батьківський об’єкт в якості композиції.

Взаємодія з усіма середовищами відбувається за допомогою таких основних методів класу Env:

- *reset()* – відновлює середовище до початкового стану для моделювання епізоду спочатку. В цьому методі можна реалізувати генерацію початкового стану в залежності від заданих параметрів для більшої уніфікації задачі.
- *step(action)* – метод, який змінює стан середовища в залежності від дій, які хоче зробити агент. Ці дії визначаються уже моделлю агента, і вони можуть бути визначені багатьма способами, такими як випадкова поведінка, запрограмована поведінка, або ж поведінка основана на нейронній мережі. Результатом виконання цього методу є такі об'єкти:
  - *observation* – спостереження які передаються агенту. Ці спостереження можуть представляти в собі зображення, що бачить агент, чи позицію у просторі окремих компонентів. Цей об'єкт також може містити в собі інформацію про поточний стан цілі.
  - *reward* – винагорода за щойно зроблену дію, використовується для оцінки ефективності роботи агента та для процесу навчання.
  - *done* – інформацію про те чи варто обновляти середовище. Залежить від того, чи кінцева точка симуляції це досягнення певної мети чи завершення епізоду
  - *info* – об'єкт з додатковою інформацією про систему.
- *render()* – метод, що відображає середовище у зрозумілому для людини форматі. Це може бути погляд зі сторонньої камери на середовище в якому рухається агент.
- *close()* - закриває середовище і усі ресурси, що були використані підчас моделювання.

Для тестування буде обрано дві різні симуляції.

- Роботизована рука на подобі людської, що обертатиме об'єкт

- Роботизований маніпулятор, що переміщуватиме предмети у просторі.

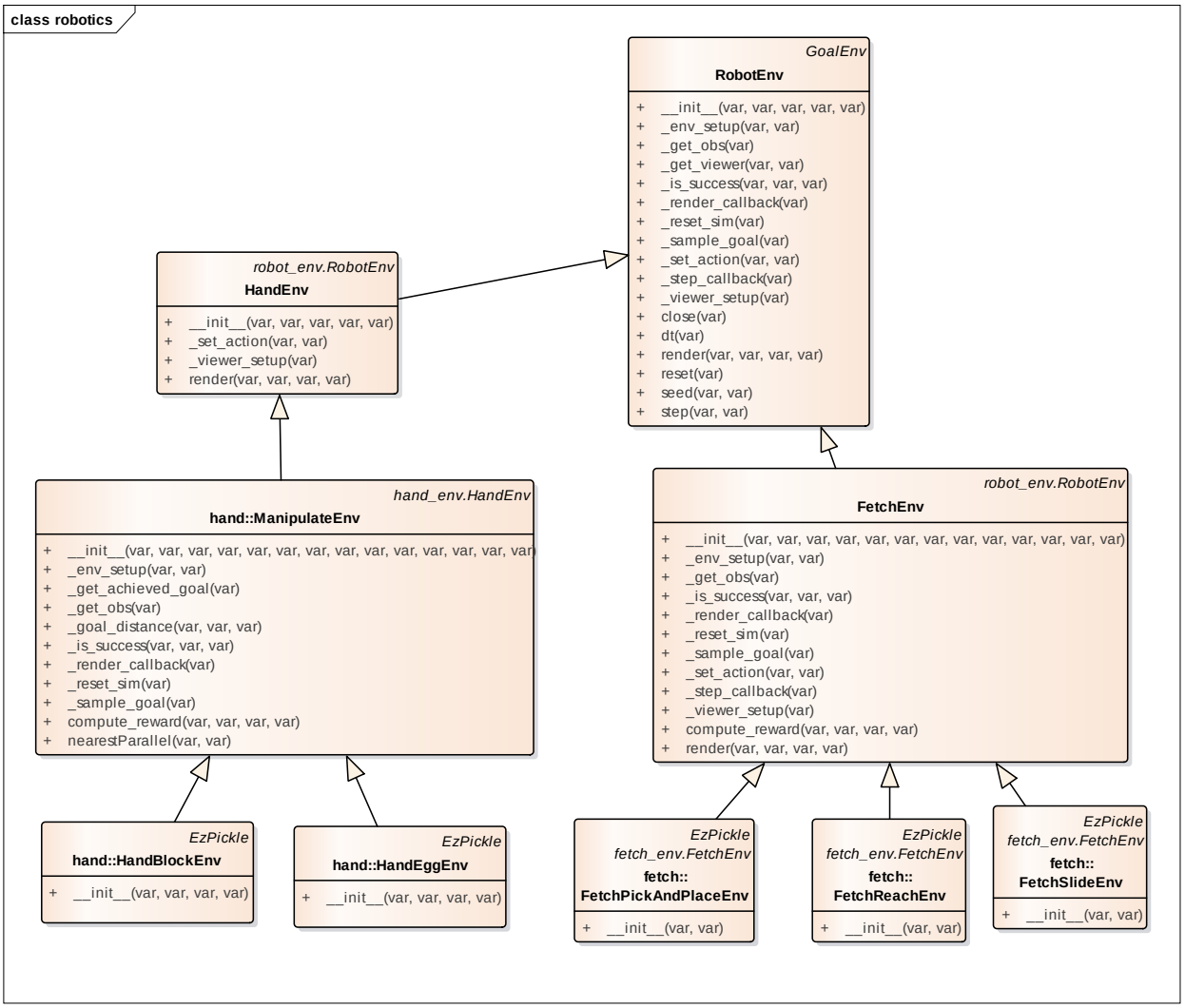

Рисунок 3.2 Діаграма класів середовищ для роботизованої руки та симулятора.

Клас *HandEnv* відповідає за середовище у якому знаходиться рука з 5ма пальцями, кожен з яких має 4 ступеня свободи та осей руху кисті. Сумарно отримуємо 24 ступеня свободи з яких 20 є повністю незалежними. Також в середині середовища знаходиться об’єкт який рука повинна обертати до певного положення. Положення об’єкта описується масивом розміру 7, 3 значення з яких це положення у 3х мірній системі координат, а решта 4 це обертання об’єкта за системою кватеріонів.

HandBlockEnv – клас що відповідає за маніпулюванням кубика з 6ма гранями.

HandEggEnv – клас що відповідає за обертання об’єкта у вигляді яйця.

Ці два об’єкти ускладняють навчання, оскільки в залежності від кута повороту рука буде взаємодіяти по-різному із об’єктами. На початку

Частина опису середовища роботизованої руки:

<body name="obj_to_manipulate" pos="1 0.87 0.2">

<geom name="obj_to_manipulate" type="box" size="0.020 0.020 0.020" material="material:obj_to_manipulate" condim="4" density="567"></geom>

<geom name="virtual_target" type="box" size="0.020 0.020 0.020" material="material:virtual_target" condim="4" contype="0" conaffinity="0" mass="0"></geom>

<site name="object:center" pos="0 0 0" rgba="1 0 0 0" size="0.01 0.01 0.01"></site>

<joint name="object:joint" type="free" damping="0.01"></joint>

</body>

Дані середовища мають різні рівні складності:

- Обертання навколо однієї з осей X Y
- Коригування обертання
- Коригування положення
- Повне обертання навколо усіх осей на будь-який кут
- Повне обертання із фіксованим положенням у просторі

Генерація цілі та початкового стану залежить від рівня складності. Приклад генерації обертання:

if self.rotation == 'z_from_any':

angle = self.np_random.choice([0, np.pi/2, np.pi, -np.pi/2])

axis = np.array([.0, .0, 1.])

target_quaterion = build_quaterion(angle, axis)

initial_parallel = self.nearestParallel (self.sim.data. get_manipulate_pos(obj_to_manipulate:join)[3:7])

target_quaterion = rotations.quat_multiply(target_quaterion, initial_parallel)

elif self.rotation == 'vertical':

angle = self.np_random.choice([0, np.pi/2, np.pi, -np.pi/2])

axis = np.array([.1, .0, 0.])

target_ quaterion = quat_from_angle_and_axis(angle, axis)

initial_parallel = self.nearestParallel (self.sim.data.get_manipulate_pos(obj_to_manipulate:joint')[3:7])

target_quaterion = rotations.quat_multiply(target_quaterion, initial_parallel)

FetchEnv відповідає за робота, який може рухати об’єкти у просторі. Його в свою чергу наслідують 3 класи з конкретними задачами:

FetchReachEnv – відповідає за переміщення головки маніпулятора.

FetchPickAndPlaceEnv – відповідає за переміщення кубика у просторі за допомогою щипців на кінці маніпулятора.

FetchSlideEnv – штовхає плоский об’єкт по слизькій поверхні.

Приклад генерації положення об’єкта на столі:

```
        if self.need_to_move:
            goal = self.robot_pos[:3] + self.np_random.uniform(-self.table_limit, self.table_limit, size=3)
            goal += self.obj_shift
            goal[2] = self.move_up
            if self.above_table and self.np_random.uniform() < 0.5:
                goal[2] += self.np_random.uniform(0, 0.45)
            else:
            goal = self.robot_pos[:3] + self.np_random.uniform(-0.15, 0.15, size=3)
        return goal.copy()
```

3.2. Реалізація алгоритму тренування

Реалізація алгоритму буде складатися із двох частин:

1. Реалізація алгоритму повтору досвіду з віртуальною ціллю HER з використанням DDPG
2. Реалізація алгоритму декомпозиції задачі на підзадачі

Реалізація розробленого алгоритму повинна відповідати усім вимогам до проектування сторонніх бібліотек для того, щоб інші дослідники могли використовувати дані розробки у своїх цілях. Для цього комунікація з компонентами алгоритму повинна проходити через єдиний інтерфейс, з максимально зрозумілими методами. Також реалізовані компоненти повинні бути відкритими для наслідування функціоналу, якщо дослідники захочуть доповнити існуючий код. Реалізація алгоритму буде відбуватися з використанням мови програмування Python. Усі операції пов’язані з нейронними мережами будуть виконуватися з використанням низькорівневих бібліотек реалізованих на мові програмування C.

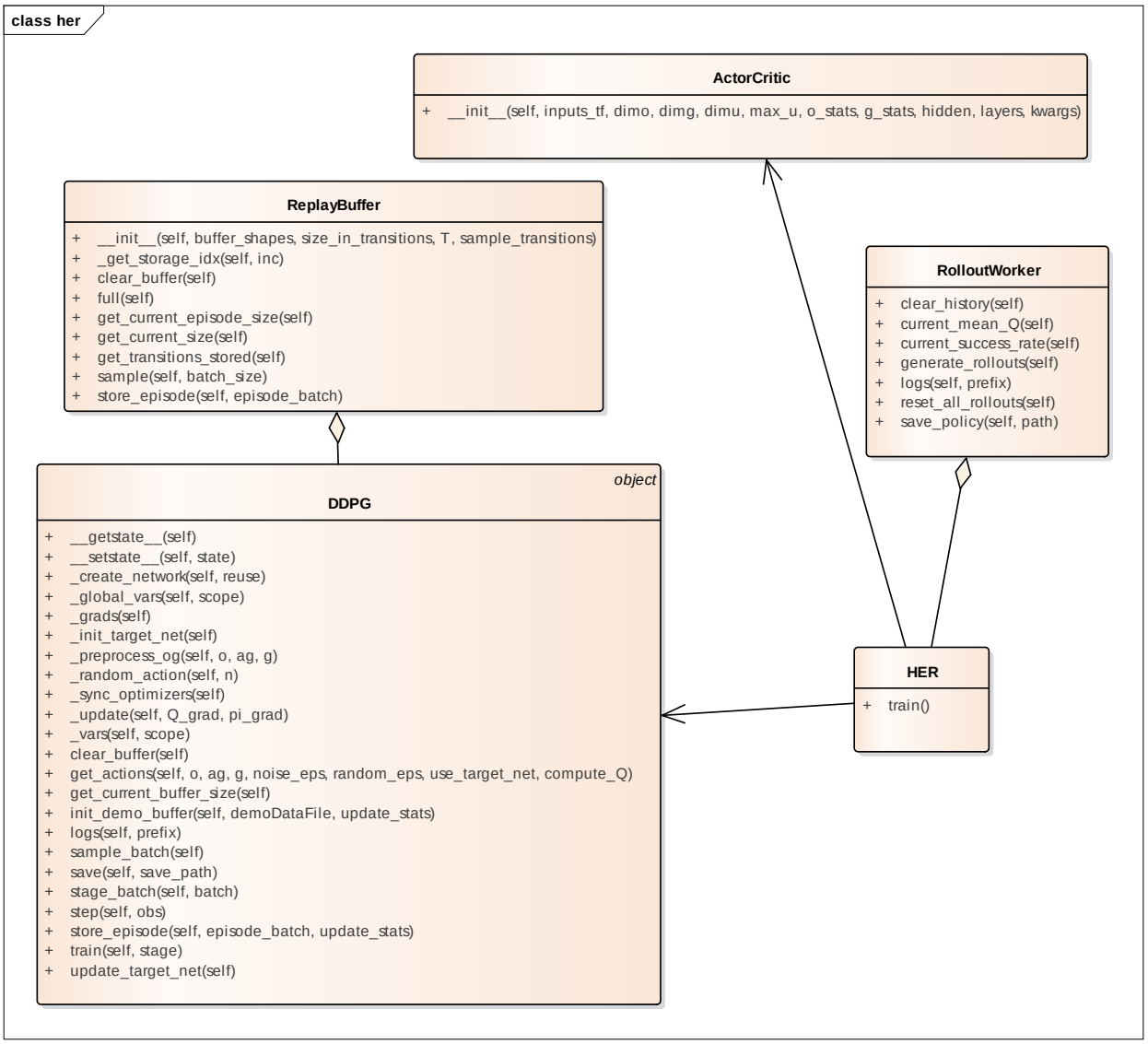

Рисунок 3.3 Діаграма класів алгоритмів HER та DDPG

Клас *DDPG* відповідає за реалізацію алгоритму DDPG у поєднанні з компонентами HER. Може працювати з використанням MPI. основні функції для взаємодії :

- *_create_network* – створює дві нейронні мережі для актора та критики в окремих контекстах Tensorflow в залежності від вхідних параметрів, деякі з них:

q_l_rate – темп навчання при зворотному поширенні помилки для мережі критики

a_l_rate - темп навчання для мережі актора

eps_n – коефіцієнт для нормалізації

hidd – кількість нейронів у прихованому шарі

hidd_l – кількість прихованих шарів

- *Функція get_actions* приймає такі параметри як спостереження, досягнену ціль, бажану ціль, параметр шуму і використовує їх в результаті вхідних параметрів для нейронної мережі, щоб визначити наступну дію, яку виконає агент
- *update_taget_net* – обновляє нейронну мережу після симуляції епізоду
- *save* – зберігає усі параметри мережі на файлову систему для подальшого відтворення

Клас Replay Buffer відповідає за збереження епізодів як були виконані під час симуляції для подальшого використання їх у навчанні. Основні функції для взаємодії :

store_episode – зберегти епізод із симуляції у буфері повтору

sample – обрати епізод із буферу для навчання в залежності від стратегії вибору.

Клас *ActorCritic* відповідає за роботу із нейронними мережами актора та критики і містить у собі код, що відповідає за навчання цих мереж використовуючи нормалізатор

Клас *RolloutWorker* відповідає за взаємодію із віртуальним середовищем та за створення епізодів для навчання.

Клас *HER* об’єднує усі ці компоненти між собою і є кінцевим інтерфейсом для комунікації із користувачем який використовуватиме даний алгоритм.

3.3. Реалізація алгоритму декомпозиції задачі для навчання

Відповідно до спроектованого плану у розділі 2 кожна необхідно, щоб агент міг навчатися кожній підзадачі окремо і незалежно від інших, тому в межах стандартного середовища буде виокремлено генерацію умов саме для окремих підзадач. На основі даних простіших середовищ буде навчатися агент з окремою нейронною мережею. Як тільки навчання завершується воно буде збережено у папку із ідентифікатором підсередовища для подальшого використання. Результатом тренування буде модель нейронної мережі та створене середовище, яке можна буде використовувати для перевірки якості навчання та демонстрації роботи над конкретною підзадачею. Модель буде збережена у окрему колекцію з ідентифікатором середовища, на якому вона навчалася.

Для об’єднання усіх моделей необхідно буде використати окреме середовище, яке буде побудоване на основі списку підзадач на які необхідно буде розбити основну задачу. Середовище виконуватиме наступні поставлені цілі:

1. Генерація початкового та кінцевого стану відповідно до основної задачі та списку переданих підзадач, що можуть використовуватись.
2. Декомпозиція основної задачі на список підзадач відповідно до переданих параметрів. Дане розбиття можливе з повторним використанням однієї і тієї ж задачі декілька разів. Наприклад перемістити маніпулятор в точку, виконати дію, перемістити маніпулятор в іншу точку.
3. Визначення меж переходів між моделями
4. Визначення яку модель використовувати для наступної дії
5. Відображення кінцевої і наступної цілі для зручності перевірки якості роботи

6. Усі інші цілі, які виконують стандартні середовища, як оцінка винагороди, чи завершено навчання і т.д.

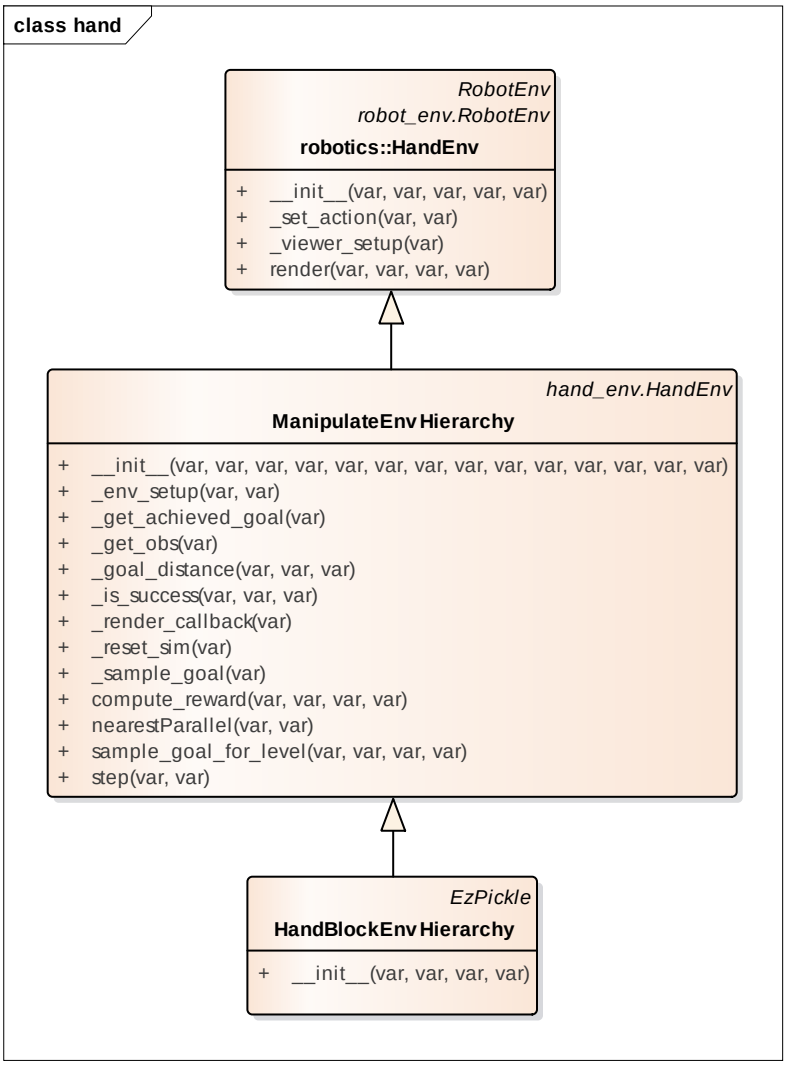

Рисунок 3.4 – Діаграма класів для роботи з декомпозованими задачами

Оскільки середовище для декомпозиції буде реалізувати схожі цілі, що й звичайне середовище, то це може бути досягнено за рахунок наслідування. На рисунку 3.4 зображено наслідування для обертання кубика. Схожа діаграма класів і для робота маніпулятора.

Клас *ManipulateEnvHierarchy* реалізує у собі декомпозицію обертання об’єкта на такі простіші задачі:

- Обертання навколо осі X
- Обертання навколо осі Z

- Коригування положення

Розбиття може використовувати підзадачі у довільному порядку довільну кількість раз. Наприклад, обертання навколо осі Y занадто складне для зафіксованої руки, тому його можна відтворити за рахунок обертання навколо осі X, потім осі Z і знову навколо осі X. На початку створення симуляції відбувається розбиття і зберігають усі етапи досягнення цілі, а також поточний етап.

Функція *nearest_parallel* знаходить найближче паралельне положення кубика.

Функція *sample_goal_for_level* визначає ціль для кожного етапу розв'язання задачі.

Функція *step* визначає чи досягена ціль для поточного етапу і якщо так, переводить середовище на наступний етап.

Агент отримує спостереження від середовища за рахунок функції *get_obs*, яке містить в собі поточний етап і в залежності від значення даного етапу агент обирає одну із збережених раніше поведінок.

Частина реалізації описаного алгоритму з використанням мови програмування Python:

```
for sub_env in extra_args['sub_envs']:
    sub_args = modify_args_for_sub_env(parsed_args, sub_env)
    model, env2 = train(sub_args, sub_extra_args)
    sub_models[sub_env] = model
    sub_envs[sub_env] = env2
```

Код вище відображає навчання окремих підзадач, наступний код показує взаємодію навчених моделей:

```
mergedEnv = build_env(known_args)
```

```
obs = mergedEnv.reset()
current_model = sub_models[extra_args['sub_envs'][0]]
while True:
    sub_actions, _, st, _ = current_model.step(observations,S=st,
    m=don)
    observations, rew, done, info = mergedEnv.step(sub_actions)
    current_model =
    sub_models[extra_args['sub_envs'][info[0]['level']]]
    total_rew += rew
    mergedEnv.render()
```

Розглянемо тепер роботу із штучними нейронними мережами з використанням бібліотеки TensorFlow. Дана бібліотека використовує два основні компоненти:

- Tensor – це набір даних зібраних у вектор або матрицю розмірністю N. Тензор зазвичай відображає внутрішню структуру нейронної мережі. Оскільки це зручно відображати нейрони кожного рівня із вагами за допомогою матриць і проводити на ними операції.

- Граф виконання – усі операції над тензором зберігаються у графі і можуть виконуватися поступово на різних фізичних юнітах. Дочірня вершина графу є результатом виконання операції надь вхідними даними.

Граф виконання може розбиватися на паралельні частини і виконуватися на багатьох процесорах одночасно, що підвищує швидкодію.

Ще однією перевагою Tensorflow є те, що можна створювати власний контекст і він буде відділений від інших, оскільки усі змінні

мають свої ідентифікатори і вони можуть бути спільними для різних ситуацій.

Для паралельного використання нейронних мереж для окремої підзадачі доцільно використовувати свій контекст із ідентифікатором envId. Таким чином ми зможемо зберігати багато схожих графів із схожими змінними. Це також доцільно для актор-критичної поведінки, оскільки ми маємо дві схожі штучні нейронні мережі, які мають оброблятися незалежно. Це має наступний вигляд

```
with tf.variable_scope(‘task_a’):
    self.internal_policy = self.create_nn_for_agent(params_a)
    new_policy = self.perform_an_update(self.internal_policy)
with tf.variable_scope('task_b'):
    self.internal_policy = self.create_nn_for_agent(params_b)
    new_policy = self.perform_an_update(self.internal_policy)
```

Навчання буде проводитися з використанням з архітектурою штучної нейронної мережі MLP. Вона є найбільш універсальною архітектурою через свою простоту, має декілька прихованих рівнів із різною кількістю нейронів.

Для навчання з підкріпленням буде обраховуватися втрата відносно очікуваної винагороди та отриманої. Для відкинення аномальних даних вони будуть поміщатися у певні межі

```
target_policy_tensor = self. target_policy_tf
    crop_range = (-self.crop_return, 0. if self.crop_pos_ else np.inf)
    clipped_tensor = tf.clip_by_value(stored_batches['reward'] + self.exp_importance * targe_policy_tensor, *cliped_tensor)
```

self.reward_loss_tensor = tf.reduce_mean(tf.square (tf.stop_gradient(clipped_tensor) - self. target_policy_tensor))

Як видно опис математичної моделі з використанням Tensorflow доволі простий і очевидний. Після обчисленої втрати необхідно оновити нейронну мережу політики поведінки. Це буде відбуватися з використанням зворотного поширення помилки. Пошук локального мінімуму це пошук похідної від певної функції і тому цей процес можна відобразити як градієнтний спуск. У актор критичної моделі дві нейронні мережі, тому потрібно їх обидві обновити.

policy_gradients_tensor = tf.gradients(self. reward_loss_tensor, self._vars('main/Actor'))

critic_gradient_policy = tf.gradients(self.critic_loss_tensor, self._vars('main/critic'))

ВИСНОВОК ДО РОЗДІЛУ 3

В даному розділі було описано реалізацію усіх елементів системи для навчання з підкріпленням з використанням мови програмування Python

Віртуальне середовище було налаштоване з використанням OpenAI gym та симулятором віртуального світу із базовими фізичними законами MuJoCo. Для тестування алгоритму було розроблено роботизовану руку із 5ма пальцями подібну до людської для обертання об’єктів та робот маніпулятор для переміщення об’єктів у просторі.

Алгоритм повторного використання досвіду із віртуальною ціллю HER був реалізований із використанням алгоритму DDPG. Програмне середовище для навчання було реалізоване на базі Tensorflow з використанням Nvidia Cuda та cuDNN.

Також було описано алгоритм декомпозиції задачі для пришвидшеного навчання. Для цього були створенні класи наслідники, що реалізують розбиття на підзадачі і генерують різні цілі для кожної із них. Агент знаючи яку саме задачу потрібно розв’язати зараз обирає спеціально натреновану модель для цього.

ДОСЛІДЖЕННЯ ТА АНАЛІЗ РЕЗУЛЬТАТІВ

4.1. Тестування роботи віртуального середовища

Як було описано у розділі 3 дослідження будуть проводитися на двох окремих віртуальних середовищах з використанням OpenAI gym. Для початку розглянемо роботизовану руку, яка обертатиме об’єкт за допомогою рухів пальців та кисті.

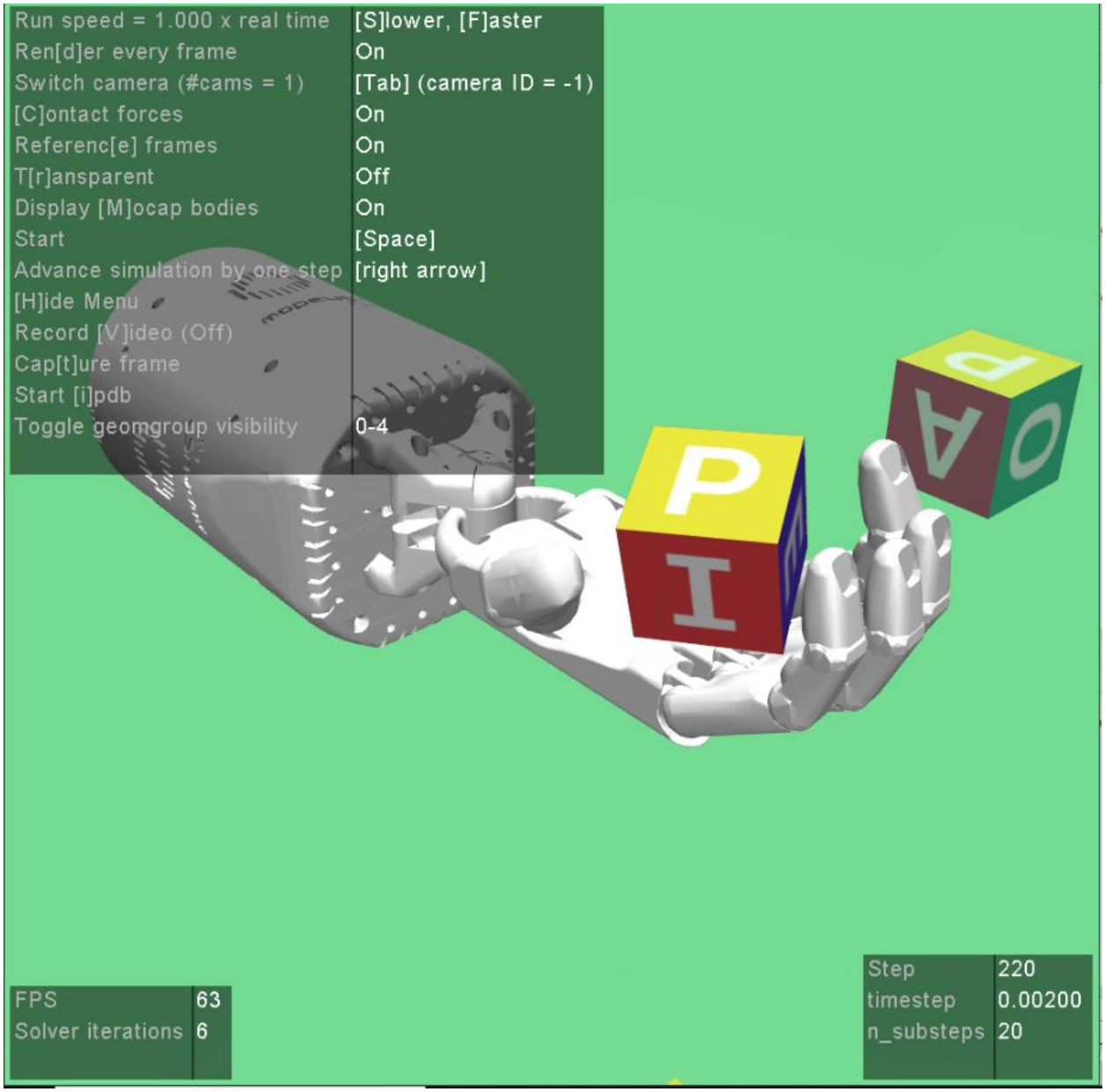

Рисунок 4.1 Роботизована рука у віртуальному середовищі OpenAI

Розроблене віртуальне середовище дозволяє відображати процес виконання задачі із обертання кубика. На рисунку 4.1 зображено один етап після декомпозиції задачі – обертання навколо осі Z.

На рисунку також зображені компоненти для зручнішого управління симуляцією, а саме:

1) Управління швидкістю симуляції(швидше при натисненні клавіші F та повільніше при натисненні клавіші S)
2) Зміна типу камери при натисненні на клавішу Tab
3) Управління камерою за допомогою мишки
4) Зупинка симуляції при натисненні на пробіл
5) Покрокове відтворення за допомогою стрілочок.

Кожен палець руки управляється окремо і має 4 незалежних ступеня свободи. Також рука може рухатися відносно кисті вліво, вправо, вверх та вниз. Сама основа руки є зафіксованою і не може рухатися, проте у майбутньому ця основа може бути приєднана до складнішого механізму.

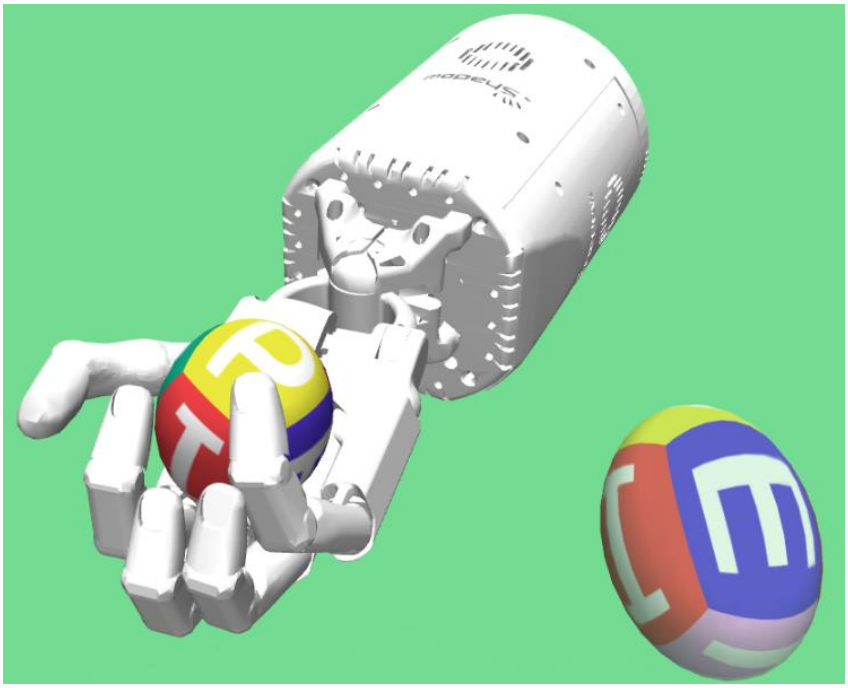

Рисунок 4.2 – обертання еліпсоїдного об’єкта

Середовище дозволяє обертати різні об’єкти, а не тільки кубик. Еліпсоїдний об’єкт є складнішим для навчання, оскільки Плече повороту у кубика більше і тому для обертання еліпсоїда потрібно більше ітерацій.

В руці знаходиться реальний фізичний об’єкт, з яким взаємодіє рука. Справа знаходиться цільове положення, до якого потрібно привести об’єкт. Можна таким чином відображати багато інформації для відслідковування правильності роботи системи.

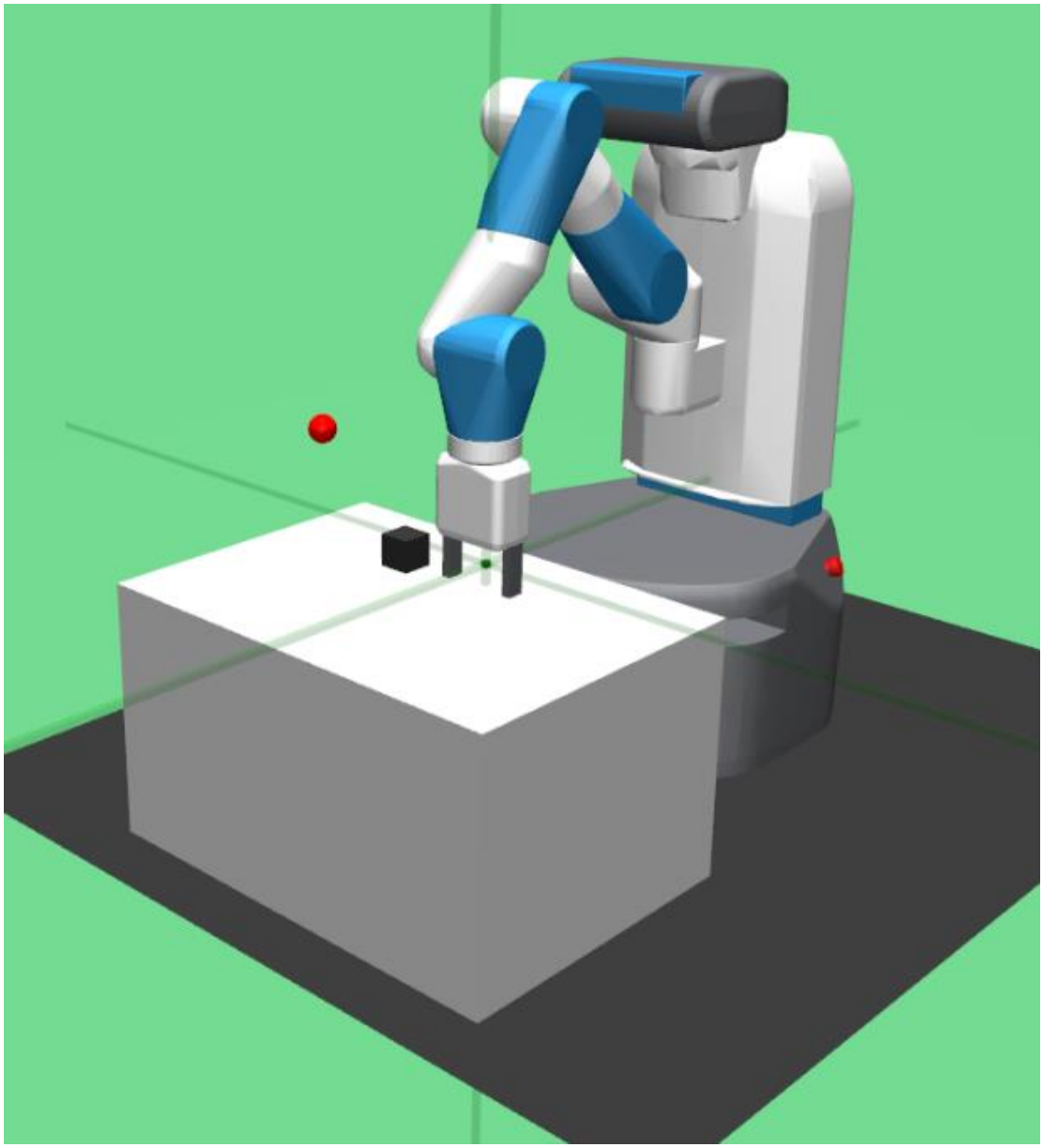

Рисунок 4.3 – робот маніпулятор.

Іншим середовищем являється робот-маніпулятор, що переміщуватиме кубик у просторі, що зображено на рисунку 4.3. Чорний кубик відображає реальний об’єкт з яким можлива взаємодія. Червона точка відображає кінцеву ціль куди необхідно перемістити кубик.

Управління маніпулятором відбувається завдяки 7ми незалежним моторам, що повертають частину маніпулятора навколо однієї осі. На голівці знаходяться щипці, що можуть зажимати об’єкт та відпускати його.

Даний тип маніпулятора активно використовується на виробництві за рахунок його універсальності. Такі типи маніпуляторів можуть виконувати різноманітні задачі, проте кожна задача вимагає довгого опису алгоритму роботи, тому дослідження на таких роботах є доволі актуальними.

4.2. Тестування розроблених алгоритмів навчання

Для початку оцінимо ефективність алгоритму повтору досвіду із віртуальною ціллю HER відносно DDPG.

Тестування проводилося на машині з наступними параметрами

- Операційна система Windows 10
- Процесор core i7 7700HQ 4 фізичних ядра
- Оперативна пам’ять 16 ГБ DDR3
- SSD диск 256 ГБ

Порівняння буде проводитися на задачі із повним обертанням кубика, оскільки дана задача має більш складніший простір можливих станів та дій. Навчання проводилося із використанням MPI і запускалося на 8ми потоках, що одночасно генерували епізоди. Для дослідження обертання кубика було обрано навчати на 150 епохах. Кожна епоха містила 50 різних епізодів. Кожен із процесів генерував окремий епізод тобто на кожну епоху приходилося: 50*8 = 400 унікальних епізодів.

Для навчання були обрані наступні параметри моделі та алгоритму HER та DDPG:

Таблиця 4.1

Параметри навчання для обертання кубика

Назва параметру	Значення	Опис
Тип нейронної мережі	MLP	Структура зв'язків нейронів у мережі
Вхідний шар	38	Положення руки(24) Положення кубика (7) Положення цілі(7)
Кількість прихованих шарів	3	Шари, що містять нейрони які будуть змінювати ваги
Розмір прихованих шарів	256	
Функція активації	ReLu	Функція, що активує нейрони
Вихідний шар	20	Список дій для окремих елементів руки
Розмір батчу	256	Кількість прикладів для однієї ітерації навчання
Імовірність випадкових дій	0,3	Для кращого дослідження середовища
Темп навчання	0,001	Швидкість знаходження локального мінімуму
Клас поведінки	Actor Critic	Дві окремі нейронні мережі
Коефіцієнт ε	0,01	Необхідно для стабілізації
Коефіцієнт γ	0,99	Фактор, що зменшує вплив майбутнього досвіду
Стратегія повтору досвіду HER	future	Алгоритм вибору епізодів для повтору

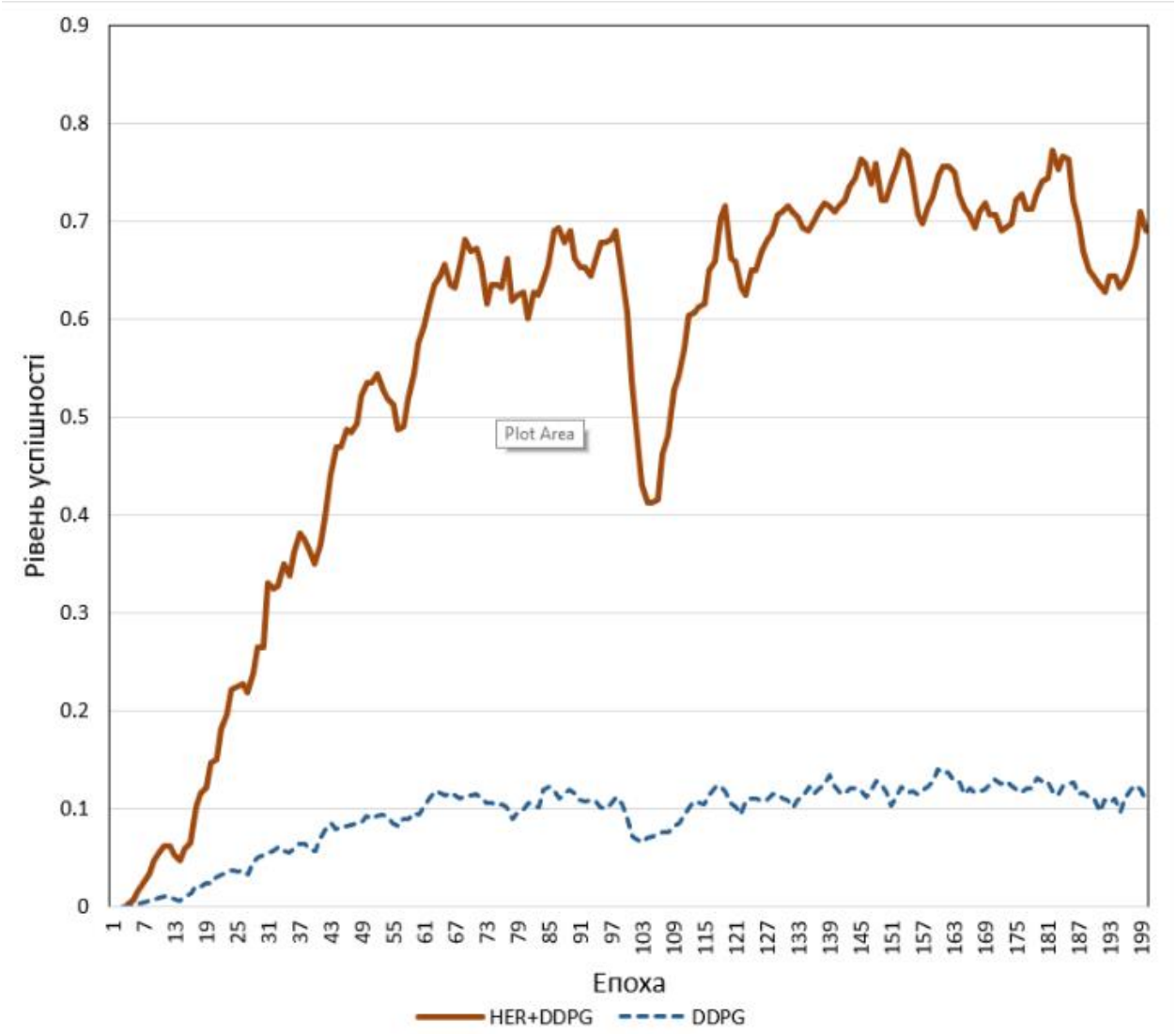

Рисунок 4.4 – Порівняння ефективності повторення досвіду з віртуальною ціллю HER відносно DDPG

Як видно з графіку на рисунку 4.4 повторення досвіду має доволі сильний позитивний вплив на успішність навчання, оскільки агент уже з перших епох отримує позитивну оцінку за віртуальну ціль і тому уже приблизно знає як себе поводити, проте з імовірністю 30% агент пробує досліджувати нову поведінку, тому він не застряє на чисто віртуальному досвіді. Як видно близько 100ї епохи було падіння успішності, проте потім це вирівнялося. Успішність DDPG 0.1% пояснюється тим, що розподіл положення кубика і цілі рівномірний і випадковий, тому інколи вони

знаходяться близько і знайти рішення простіше ніж 3 оберти навколо різних осей.

Тепер порівняємо процес навчання при різній кількості паралельних обробників.

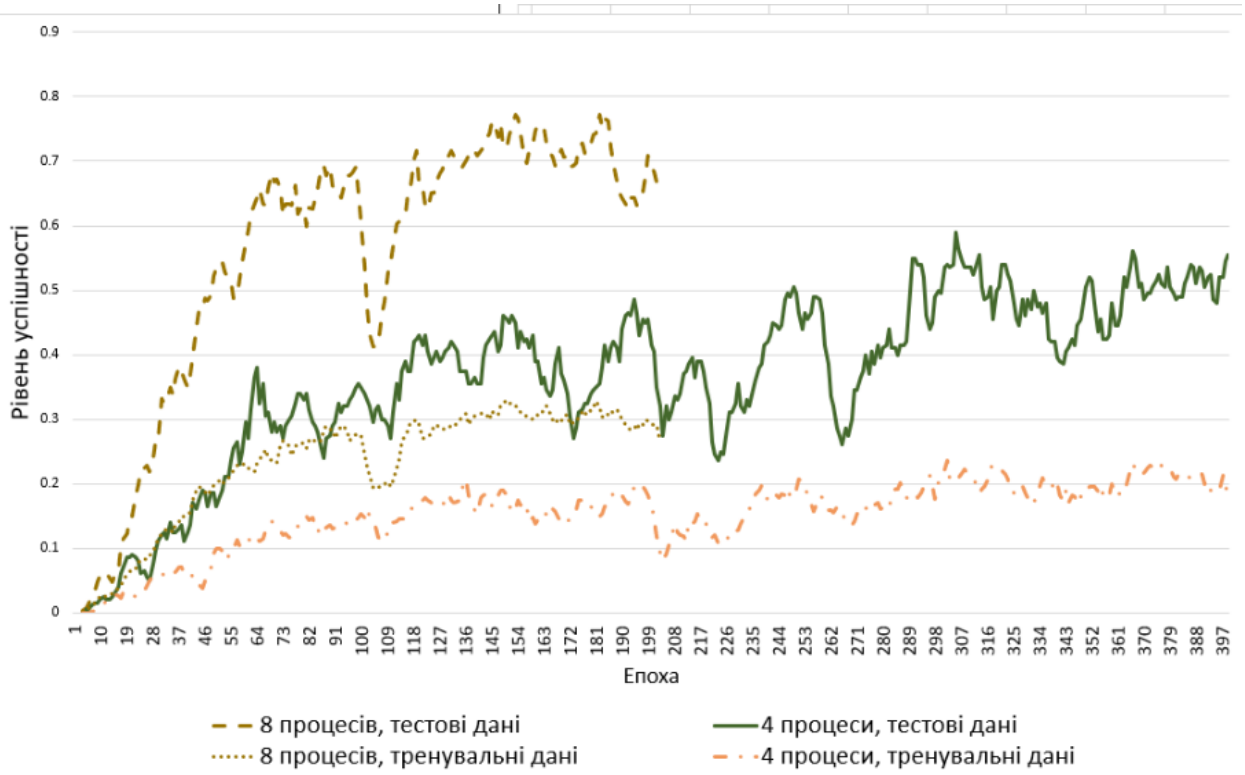

Рисунок 4.5 – Залежність успішності виконання від кількості паралельних обробників.

Цього разу тренування на 4х паралельних обробниках проводилося із кількістю епох – 400, оскільки за 400 епох обробники згенерують таку ж саму кількість унікальних епізодів. Після 200 епох для 8ми процесів рівень успішності уже не покращувався, тому максимальна успішність складає приблизно 75%. Для 4х процесів рівень успішності практично не покращувався після 400 епох, тому максимальна успішність складає приблизно 55%, що є нижчою від очікуваної. Це пояснюється тим, що важливо генерувати більше унікальних ситуацій на перших ітераціях, оскільки далі вплив нового досвіду зменшується.

Також цікаво спостерігати, що рівень успішності при навчанні є значно нижчим, ніж рівень успішності при перевірці на тестових даних. Це пояснюється специфікою роботи алгоритму HER. Оскільки під час навчання у буфер повтору попадають як негативний досвід, так і позитивний досвід із віртуальною ціллю. А під час тестування нейронна мережа уже може давати позитивні результати навіть на тих випадках, які не були згенеровані, проте були створенні віртуально самим алгоритмом.

Тепер перевіримо ефективність навчання при використанні декомпозиції задачі. В першу чергу розглянемо обертання кубика. Повна задача заключається у обертанні кубика рукою навколо будь-яких осей на будь-який кут в межах повного оберту. Для тестування було вирішено розбити на такі етапи задачу:

- обертання навколо осі Z
- обертання навколо осі X
- коригування положення відповідно до кінцевої цілі

Під час роботи алгоритму, що об’єднує навчені нейронні мережі перші дві цілі можуть виконуватися декілька разів, щоб відобразити обертання навколо осі Y, оскільки розроблена модель руки не призначена для маніпулювання навколо осі Y.

Для усіх трьох підзадач буде використовуватись однакові параметри мережі та алгоритмів, а саме навчання буде проходити протягом 100 епох. Для порівняння візьмемо ефективність розв’язання повної задачі без декомпозиції. Порівнювати планується кількість ітерацій після яких успішність уже практично не підвищується та фінальну успішність. Також інтерес викликає яка саме поведінка буде у агента без декомпозиції. Це необхідно для того щоб зрозуміти який ефект приносить декомпозиція, оскільки вона може бути не достатньо оптимальною і агент виконуватиме задачу, проте за довший час.

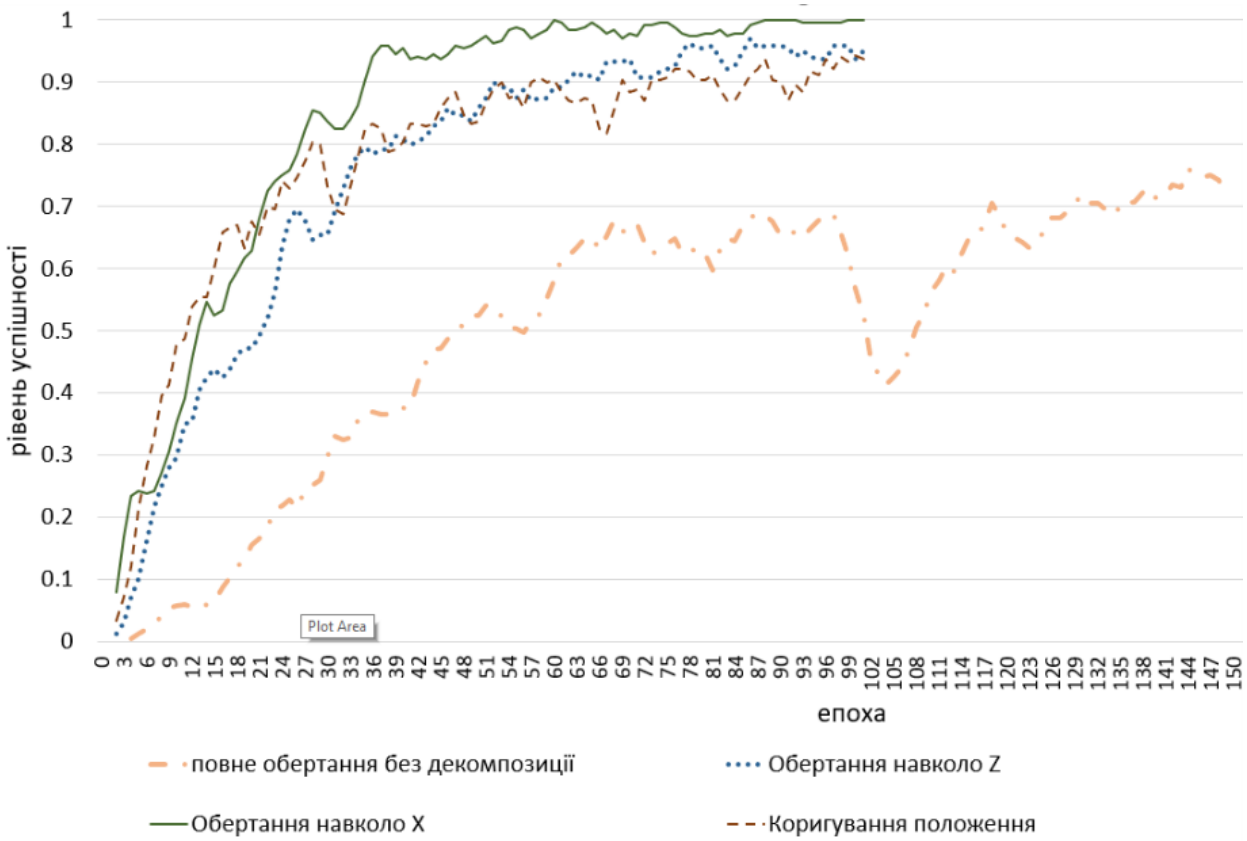

Рисунок 4.6 порівняння ефективності навчання декомпозованих задач для обертання кубика із оригінальною

Оригінальна задача навчалася протягом 200 епох, проте на графіку зображено лише 150, бо далі тенденція не змінювалася. Як видно на рисунку 4.6 ефективність навчання для декомпозованих значно краща. 70% успішності вони досягли на 20-30й епосі в той час як оригінальна лише після 130ї. Також важливо звернути увагу на рівень успішності для декомпозованих задач кінцевий рівень успішності складає 95-98% у той час, як у оригінальної лише ~75%. Тепер оцінимо сумарну успішність. Теоретично вона має складати біля 93%, оскільки похибки усіх рівнів множаться, проте не усі рівні можуть бути задіяні. Після тестування на 10000 епізодах алгоритму середня успішність склала 78%, що нижче очікуваної. Після аналізу було виявлено, що інколи нейронна мережа просто не знає, що робити у конкретному стані, бо положення кубика змістилося відносно очікуваного кінцевого положення попередньої цілі, а

навчання проводилося з очікуванням, що кубик для наступної цілі буде розташований у точній позиції для наступної цілі. Тому було проведено ще одне навчання, де до початкової позиції кубика для кожної із підзадач було добавлене незначне зміщення. В результаті сумарна ефективність роботи усіх компонентів склала 91%, що значно ближче до очікуваної успішності.

Рисунок 4.7 демонстрація роботи алгоритму із декомпозицію задачі, початковий стан

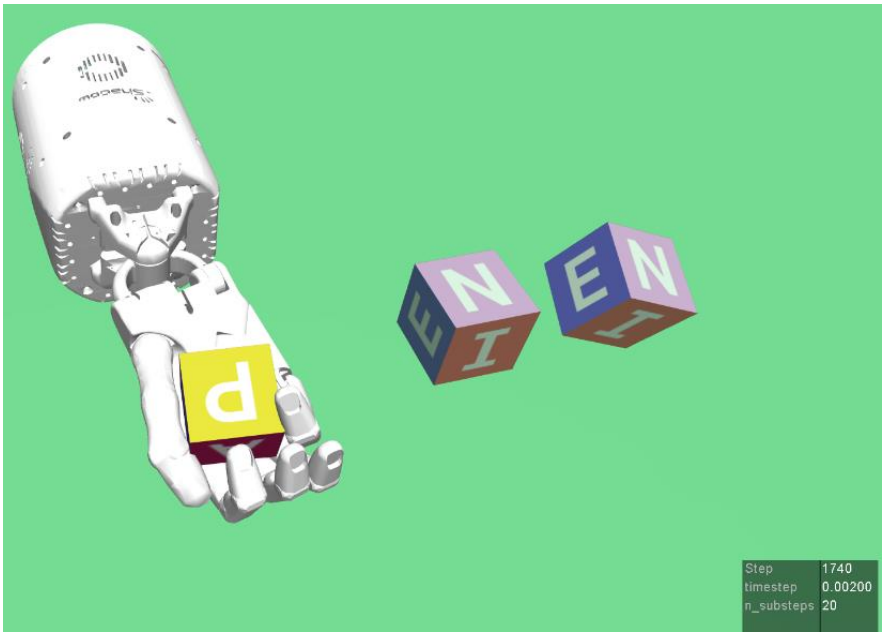

Рисунок 4.8 - демонстрація роботи алгоритму, виконана перша ціль

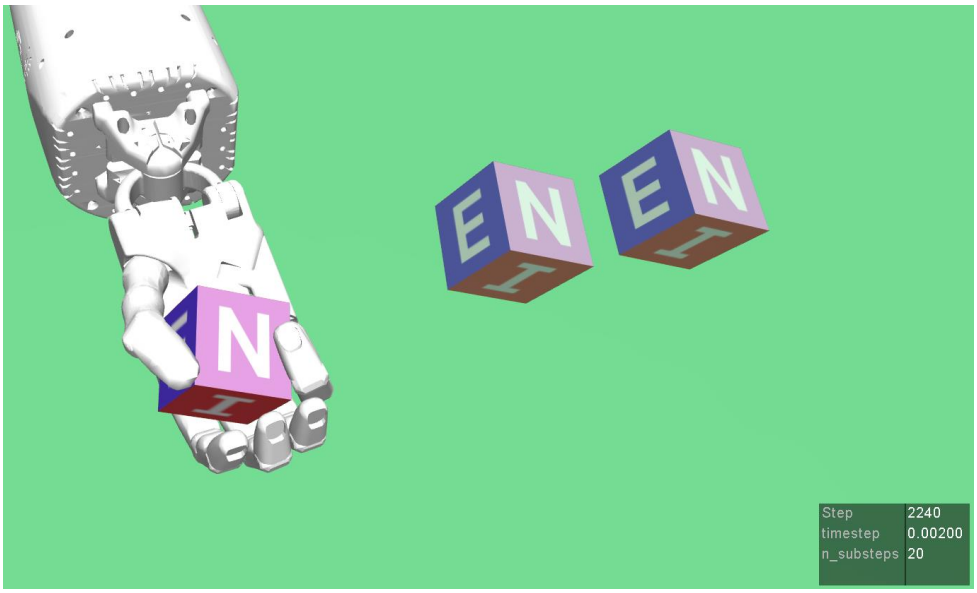

Рисунок 4.9 - демонстрація роботи алгоритму, виконана фінальна ціль

Як бачимо з рисунків 4.7, 4.8, 4.9 алгоритм працює правильно, де усі три натреновані мережі були активовано у правильному порядку, тому була досягнута кінцева ціль.

Для демонстрації роботи алгоритму наочно було обрано зображати кінцеву ціль поряд із поточною, щоб бачити динаміку активації окремої мережі. Після аналізу поведінки при навчанні з декомпозицією та без було виявлено, що без декомпозиції агент знайшов шляхи не описані в декомпозиції, але в переважній більшості випадків робив ту ж послідовність, що і агент з декомпозицією, проте з певним зміщенням осей. Це пояснюється специфікою будови руки, якій зручніше обертати кубик навколо осей прив’язаних до положення пальців, а не кисті.

Порівняємо також роботу алгоритму на іншому середовищі, роботі маніпуляторі, що переміщає об’єкти у просторі. Для розбиття достатньо використати дві підзадачі:

- Переміщення маніпулятора з будь-якої точки у будь-яку іншу, точки мають бути в межах досягнення робота.
- Фіксація об’єкта затискачами.

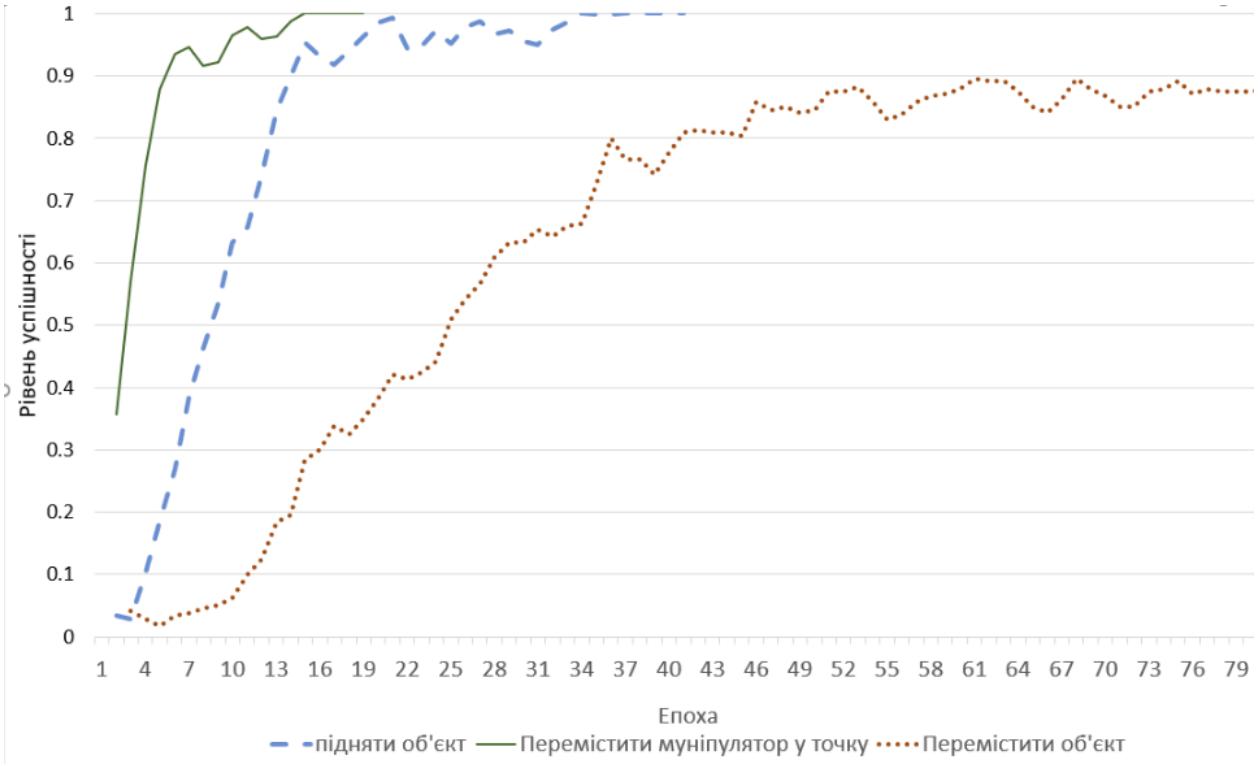

Рисунок 4.10 - порівняння ефективності навчання декомпозованих задач з оригінальною, для робота маніпулятора

Після навчання, на графіку зображеному на рисунку 4.10 бачимо кращу швидкість та якість навчання для вирішення задачі переміщення об’єкта у просторі роботом маніпулятором. Кінцева успішність для підзадач склала ~98% у той час, як для оригінальної близько 88%. Після перевірки об’єднання підзадач фінальна середня успішність склала 97% на 10000 епізодах.

Тепер розглянемо процес роботи маніпулятора у режимі реального часу. Для зручності відображення, будемо зображати кожну наступну ціль фіолетовим маркером, а кінцеву червоним. Із розбиття задачі положення фіолетового маркера буде двічі над кубиком, оскільки першим етапом буде розмістити щипці над кубиком, а другим його просто підняти. Це необхідно навіть, якщо для кінцевої цілі кубик має бути на столі, щоб не пошкоджувати стіл і об’єкт у реальній роботі.

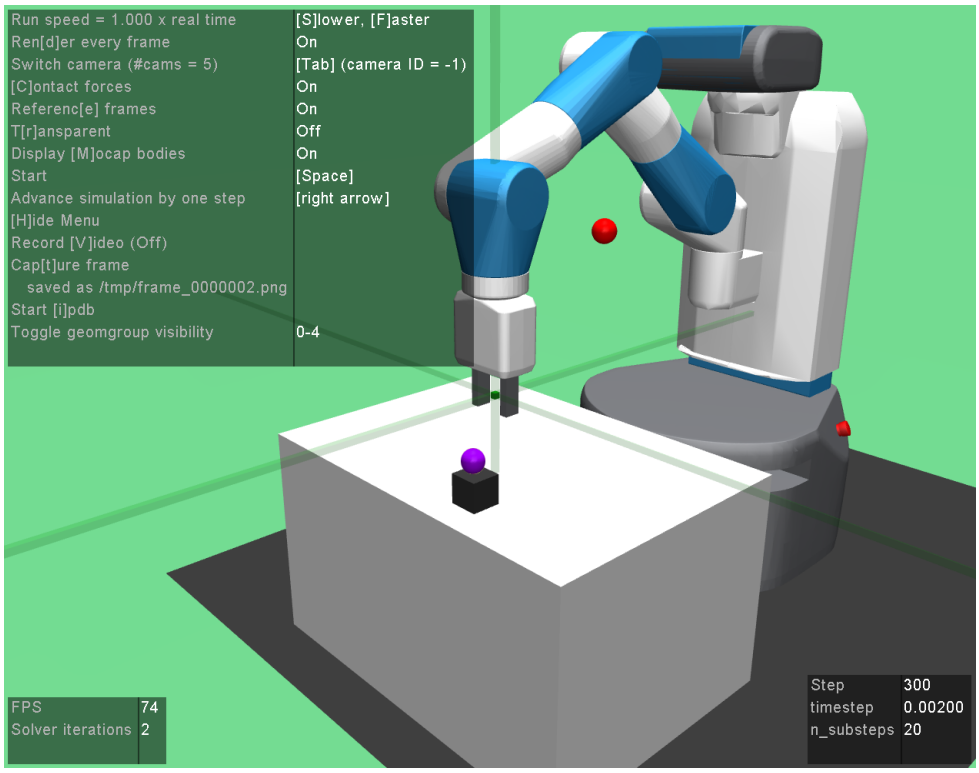

Рисунок 4.11 – приклад роботи робота маніпулятора з декомпозованою задачею, початковий стан

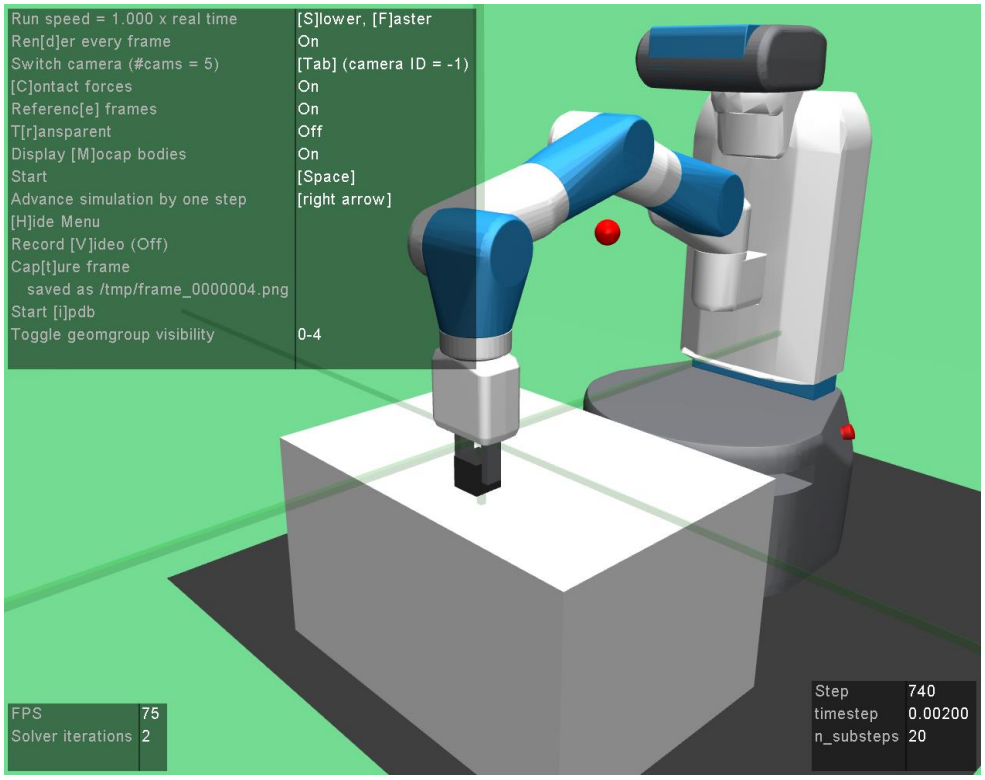

Рисунок 4.12 - приклад роботи робота маніпулятора з декомпозованою задачею, середній стан

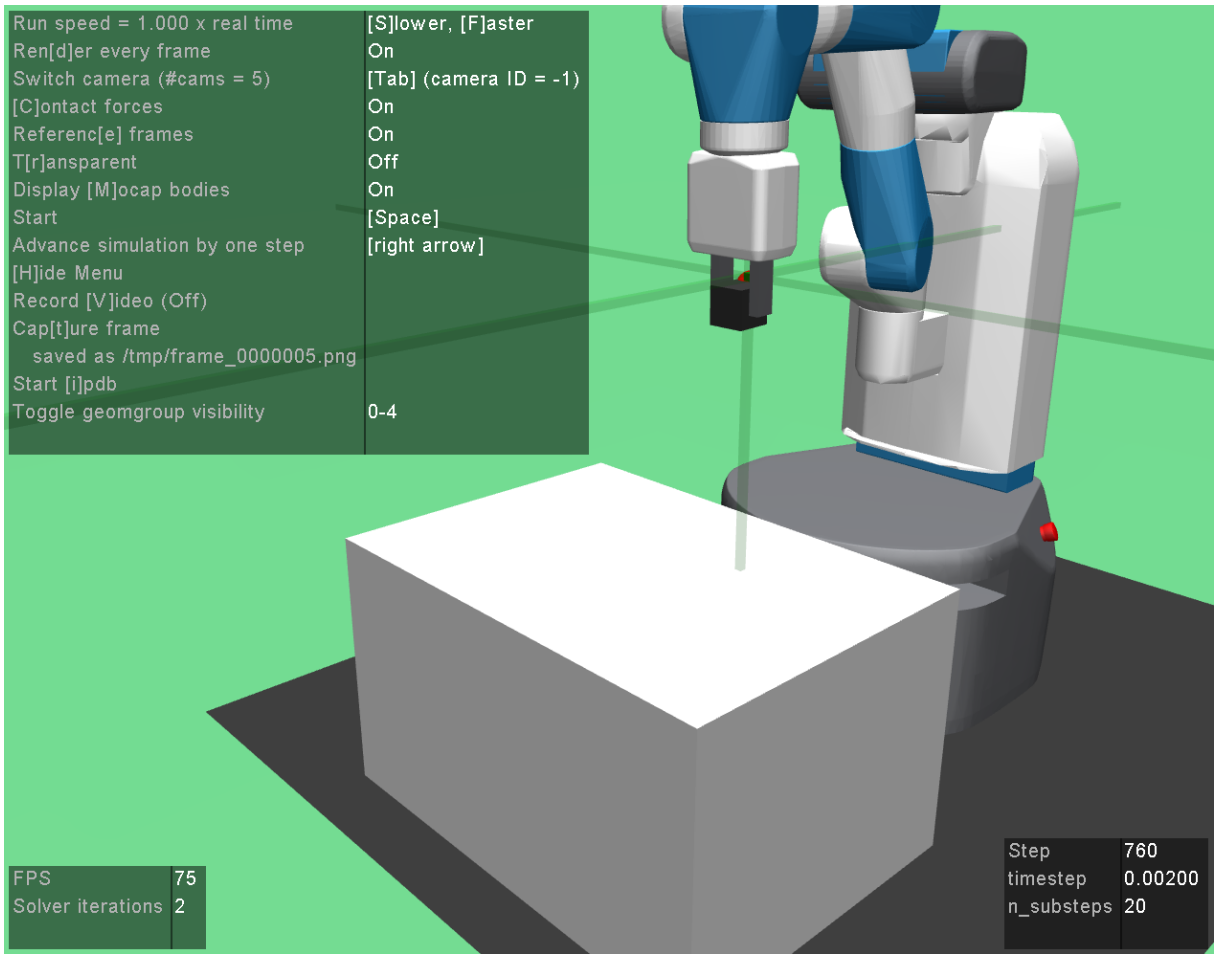

Рисунок 4.13 - приклад роботи робота маніпулятора з декомпозованою задачею, ціль виконана.

Як видно із рисунків 4.11-4.13 робот виконує дії правильно у правильній послідовності. Оскільки для навчання дозволялося певна похибка, то її можна помітити і у результатах. Цікавим спостереженням є те, що на досягнення цілі роботу необхідно всього 20-40 часових проміжків, а для того щоб підняти аж 300, що пояснюється складністю фізичної взаємодії.

ВИСНОВОК ДО РОЗДІЛУ 4

В даному розділі було проведено тестування розробленої системи для навчання роботизованих агентів. Була проведена демонстрація роботи віртуального середовища на базі OpenAI gym на таких моделях:

- Роботизована рука на подобі людської, що має 24 ступеня свободи, задачею якої є обертання кубика
- Робот-маніпулятор із 7ма ступенями свободи, що переміщає об’єкт у просторі з використанням затискачів.

Далі було проведене тестування ефективності роботи розроблених алгоритмів на описаних вище середовищах. Спочатку було проведено тренування з методикою повторення досвіду із віртуальною ціллю та без неї. В результаті було наведено, що дана методика значно покращує якість навчання(75% проти 10%). Також була показана специфіка роботи даного алгоритму при навчанні.

Після цього було проведено тестування ефективності декомпозиції задачі, що показало користь використання даної методики для швидшого та ефективнішого навчання. Нейронні мережі навчалися швидше та показували кращі результати на підзадачах і навіть після об’єднання коефіцієнт успішності був вищим у алгоритму з декомпозованими задачами, аніж у алгоритму, де рішення знаходилося без розділення. Для руки, що обертає кубик це склало 91% відносно 75%, а у робота-маніпулятора 97% відносно 88%. А саме навчання займало вдвічі, а то і втричі менше часу.

ВИСНОВКИ

В ході роботи на магістерською дисертацією був розроблений метод що дозволив підвищити швидкість навчання і покращив рівень успішності для задач з використанням роботизованих агентів.

У першому розділі були розглянуті існуючи методи навчання з підкріпленням, проте було виявлено, що жоден із методів не дозволяє агенту ефективно навчатися при доволі великій множині можливих станів середовища і великій множині можливих одночасних дій агента. Тому було поставлено наступні цілі:

- Розробити метод, який зменшить кількість ітерацій навчання для досягнення тієї ж успішності, що й існуючі алгоритми
- Розробити метод, який підвищить успішність навчання у порівнянні з існуючими
- Розроблений метод повинен масштабуватися на ще більш складні середовища і задачі.
- Розробити віртуальне середовище для тестування роботи розробленого методу.

У другому розділі було запропоновано використовувати поєднання двох методик. Перша це навчання з повторенням досвіду із віртуальною ціллю. Даних підхід дозволив пришвидшити процес навчання, оскільки агент швидше розумів як потрібно себе поводити. Інший метод заключався у декомпозиції задачі і тому дозволив навчатися складній задачі швидше і з більшим рівнем успішності, оскільки навчання підзадачам проводилося паралельно.

У третьому розділі була описана реалізація усіх спроектованих компонентів із використанням мови програмування Python. Робота з глибоким навчанням проводилася за допомогою Tensorflow та Nvidia

cudaDNN. Також була описана реалізація віртуального середовища на основі OpenAI gym.

У четвертому розділі було проведено дослідження розроблених алгоритмів на віртуальному середовищі на двох типах задач

- Роботизована рука, подібна до людської, повинна обертати кубик навколо усіх осей на довільних кут
- Робот маніпулятор, що переміщає об’єкти у просторі

Під час тестування було виявлено, що кількість ітерацій необхідна для розв’язання цих задач значно менше, приблизно вдвічі, а успішність виконання зросла на 10-15%.

В процесі роботи над магістерською дисертацією були систематизовані та засвоєні абсолютно нові теоретичні та практичні навички для роботи із глибоким машинним навчанням з підкріпленням.

В подальших дослідженнях планується знайти спосіб автоматичної, більш оптимальної декомпозиції задачі, якій агент самостійно буде навчатися за рахунок кластеризації своїх дій та отриманого досвіду. Також планується дослідити спосіб автоматичного визначення яку модель поведінки обрати в залежності від умов середовища.

СПИСОК ВИКОРИСТАНОЇ ЛІТЕРАТУРИ

1. Lex Fridman MIT Deep Learning Introduction [Електронний ресурс] – Режим доступу до ресурсу: https://deeplearning.mit.edu/
2. Hindsight Experience Replay / Andrychowicz M., Wolski F., Ray A., Schneider J. та інші. – arXiv, 2017. – 15 с. – (препринт arXiv: 1707.01495)
3. Multi-Goal Reinforcement Learning: Challenging Robotics Environments and Request for Research / Plappert M., Andrychowicz M., Ray A., McGrew B. та інші. – arXiv, 2018. – 16 с. – (препринт arXiv: 1802.09464)
4. Репозиторій OpenAI Baselines[Електронний ресурс] – Режим доступу до ресурсу : https://github.com/openai/baselines.
5. Levy A. Hierarchical actor-critic./ Levy A., Platt R., Saenko K. – arXiv, 2017 – 16 с. – (препринт arXiv:1712.00948).
6. Abadi M. Tensorflow: Large-scale machine learning on heterogeneous / Abadi M., AgarwalA., Barham P., Brevdo E – arXiv, 2016 – 19 с. – (препринт arXiv:1603.04467).
7. Репозиторій OpenAI gym [Електронний ресурс] – Режим доступу до ресурсу : https://github.com/openai/gym
8. Mnih V. Human-level control through deep reinforcement learning. / Mnih V., Kavukcuoglu K., Silver D., Rusu A. - Nature (2015) – номер 518(7540) – С. 529–533.
9. Stable baselines, реалізація основ алгоритмів для навчання з підкріпленням [Електронний ресурс] – Режим доступу до ресурсу: https://stable-baselines.readthedocs.io/
10. Lillicrap T. P. Continuous control with deep reinforcement learning Lillicrap / T. P., Hunt J. J., Pritzel A., Heess N – arXiv, 2015 – 14 с. – (препринт arXiv:1509.02971)

ДОДАТОК А ЛІСТИНГ КОДУ

```
import sys
import re
import multiprocessing
import os.path as osp
import gym
from collections import defaultdict
import tensorflow as tf
import numpy as np

from baselines.common.vec_env import VecFrameStack, VecNormalize, VecEnv
from baselines.common.vec_env.vec_video_recorder import VecVideoRecorder
from baselines.common.cmd_util import common_arg_parser, parse_unknown_args,
make_vec_env, make_env
from baselines.common.tf_util import get_session
from baselines import logger
from importlib import import_module


try:
    from mpi4py import MPI
except ImportError:
    MPI = None

try:
    import pybullet_envs
except ImportError:
    pybullet_envs = None

try:
    import roboschool
except ImportError:
    roboschool = None

_game_envs = defaultdict(set)
for env in gym.envs.registry.all():
    # TODO: solve this with regexes
    env_type = env.entry_point.split(':')[0].split('.')[-1]
    _game_envs[env_type].add(env.id)


def train(args, extra_args):
    env_type, env_id = get_env_type(args)
    print('env_type: {}'.format(env_type))

    total_timesteps = int(args.num_timesteps)
    seed = args.seed

    learn = get_learn_function(args.alg)
    alg_kwargs = get_learn_function_defaults(args.alg, env_type)
    alg_kwargs.update(extra_args)

    env = build_env(args)
    if args.save_video_interval != 0:
        env = VecVideoRecorder(env, osp.join(logger.get_dir(), "videos"),
record_video_trigger=lambda x: x % args.save_video_interval == 0,
video_length=args.save_video_length)
```

```
    if args.network:
        alg_kwargs['network'] = args.network
    else:
        if alg_kwargs.get('network') is None:
            alg_kwargs['network'] = get_default_network(env_type)

    print('Training {} on {}:{} with arguments \n{}'.format(args.alg, env_type,
env_id, alg_kwargs))

    model = learn(
        env=env,
        seed=seed,
        total_timesteps=total_timesteps,
        **alg_kwargs
    )

    return model, env


def build_env(args):
    ncpu = multiprocessing.cpu_count()
    if sys.platform == 'darwin': ncpu //= 2
    nenv = args.num_env or ncpu
    alg = args.alg
    seed = args.seed

    env_type, env_id = get_env_type(args)

    if env_type in {'atari', 'retro'}:
        if alg == 'deepq':
            env = make_env(env_id, env_type, seed=seed,
wrapper_kwargs={'frame_stack': True})
        elif alg == 'trpo_mpi':
            env = make_env(env_id, env_type, seed=seed)
        else:
            frame_stack_size = 4
            env = make_vec_env(env_id, env_type, nenv, seed,
gamestate=args.gamestate, reward_scale=args.reward_scale)
            env = VecFrameStack(env, frame_stack_size)

    else:
        config = tf.ConfigProto(allow_soft_placement=True,
                                intra_op_parallelism_threads=1,
                                inter_op_parallelism_threads=1)
        config.gpu_options.allow_growth = True
        get_session(config=config)

        flatten_dict_observations = alg not in {'her'}
        env = make_vec_env(env_id, env_type, args.num_env or 1, seed,
reward_scale=args.reward_scale,
flatten_dict_observations=flatten_dict_observations)

        if env_type == 'mujoco':
            env = VecNormalize(env, use_tf=True)

    return env


def get_env_type(args):
```

```
    env_id = args.env

    if args.env_type is not None:
        return args.env_type, env_id

    # Re-parse the gym registry, since we could have new envs since last time.
    for env in gym.envs.registry.all():
        env_type = env.entry_point.split(':')[0].split('.')[-1]
        _game_envs[env_type].add(env.id)  # This is a set so add is idempotent

    if env_id in _game_envs.keys():
        env_type = env_id
        env_id = [g for g in _game_envs[env_type]][0]
    else:
        env_type = None
        for g, e in _game_envs.items():
            if env_id in e:
                env_type = g
                break
        if ':' in env_id:
            env_type = re.sub(r':.*', '', env_id)
        assert env_type is not None, 'env_id {} is not recognized in env
types'.format(env_id, _game_envs.keys())

    return env_type, env_id


def get_default_network(env_type):
    if env_type in {'atari', 'retro'}:
        return 'cnn'
    else:
        return 'mlp'


def get_alg_module(alg, submodule=None):
    submodule = submodule or alg
    try:
        # first try to import the alg module from baselines
        alg_module = import_module('.'.join(['baselines', alg, submodule]))
    except ImportError:
        # then from rl_algs
        alg_module = import_module('.'.join(['rl_' + 'algs', alg, submodule]))

    return alg_module


def get_learn_function(alg):
    return get_alg_module(alg).learn


def get_learn_function_defaults(alg, env_type):
    try:
        alg_defaults = get_alg_module(alg, 'defaults')
        kwargs = getattr(alg_defaults, env_type)()
    except (ImportError, AttributeError):
        kwargs = {}
    return kwargs
```

```
def parse_cmdline_kwargs(args):
    '''
    convert a list of '='-spaced command-line arguments to a dictionary,
evaluating python objects when possible
    '''
    def parse(v):

        assert isinstance(v, str)
        try:
            return eval(v)
        except (NameError, SyntaxError):
            return v

    return {k: parse(v) for k,v in parse_unknown_args(args).items()}


def modify_args_for_sub_env(args, sub_env):
    args.env += sub_env + '-v0'
    if args.save_path is not None:
        args.save_path += sub_env
    if args.log_path is not None:
        args.log_path += sub_env
    return args


def modify_extra_args(args, sub_env):
    if 'load_path' in args:
        args['load_path'] += sub_env
    return args


def configure_logger(log_path, **kwargs):
    if log_path is not None:
        logger.configure(log_path)
    else:
        logger.configure(**kwargs)


def main(args):
    # configure logger, disable logging in child MPI processes (with rank > 0)

    """
    --sub_envs=['ZFromAny','Vertical','NearestXYZ']
    """
    arg_parser = common_arg_parser()
    known_args, unknown_args = arg_parser.parse_known_args(args)
    extra_args = parse_cmdline_kwargs(unknown_args)

    sub_models = dict()
    sub_envs = dict()

    for sub_env in extra_args['sub_envs']:
        parsed_args, sub_extra_args = arg_parser.parse_known_args(args)
        sub_extra_args = parse_cmdline_kwargs(sub_extra_args)
        sub_args = modify_args_for_sub_env(parsed_args, sub_env)
        sub_extra_args = modify_extra_args(sub_extra_args, sub_env)

        if MPI is None or MPI.COMM_WORLD.Get_rank() == 0:
            rank = 0
```

```
            configure_logger(sub_args.log_path)
        else:
            rank = MPI.COMM_WORLD.Get_rank()
            configure_logger(sub_args.log_path, format_strs=[])

        model, env2 = train(sub_args, sub_extra_args)
        env2.close()

        if sub_args.save_path is not None and rank == 0:
            save_path = osp.expanduser(sub_args.save_path)
            model.save(save_path)

        sub_models[sub_env] = model
        sub_envs[sub_env] = env2

    modify_args_for_sub_env(known_args, '')
    mergedEnv = build_env(known_args)
        # sub_envs[extra_args['sub_envs'][0]]

    if parsed_args.play:
        logger.log("Running trained model")
        obs = mergedEnv.reset()
        current_model = sub_models[extra_args['sub_envs'][0]]
        state = current_model.initial_state if hasattr(current_model,
'initial_state') else None
        dones = np.zeros((1,))

        episode_rew = np.zeros(mergedEnv.num_envs) if isinstance(mergedEnv,
VecEnv) else np.zeros(1)
        while True:
            if state is not None:
                actions, _, state, _ = current_model.step(obs,S=state, M=dones)
            else:
                actions, _, _, _ = current_model.step(obs)

            obs, rew, done, info = mergedEnv.step(actions)
            current_model = sub_models[extra_args['sub_envs'][info[0]['level']]]
            episode_rew += rew
            mergedEnv.render()
            done_any = done.any() if isinstance(done, np.ndarray) else done
            if done_any:
                for i in np.nonzero(done)[0]:
                    print('episode_rew={}'.format(episode_rew[i]))
                    episode_rew[i] = 0

    mergedEnv.close()

if __name__ == '__main__':
    main(sys.argv)
import os
import numpy as np

from gym import utils, error
from gym.envs.robotics import rotations, hand_env
from gym.envs.robotics.utils import robot_get_obs

try:
    import mujoco_py
except ImportError as e:
```

```
    raise error.DependencyNotInstalled("{}. (HINT: you need to install mujoco_py,
and also perform the setup instructions here: https://github.com/openai/mujoco-
py/.)".format(e))


def quat_from_angle_and_axis(angle, axis):
    assert axis.shape == (3,)
    axis /= np.linalg.norm(axis)
    quat = np.concatenate([[np.cos(angle / 2.)], np.sin(angle / 2.) * axis])
    quat /= np.linalg.norm(quat)
    return quat


# Ensure we get the path separator correct on windows
MANIPULATE_BLOCK_XML = os.path.join('hand', 'manipulate_block.xml')
MANIPULATE_EGG_XML = os.path.join('hand', 'manipulate_egg.xml')
MANIPULATE_PEN_XML = os.path.join('hand', 'manipulate_pen.xml')


class ManipulateEnv(hand_env.HandEnv):
    def __init__(
        self, model_path, target_positions, target_rotations,
        target_position_range, reward_type, initial_qpos=None,
        randomize_initial_position=True, randomize_initial_rotation=True,
        distance_threshold=0.01, rotation_threshold=0.1, n_substeps=20,
relative_control=False,
        ignore_z_target_rotation=False,
    ):
        self.target_positions = target_positions
        self.target_rotations = target_rotations
        self.target_position_range = target_position_range
        self.parallel_quats = [rotations.euler2quat(r) for r in
rotations.get_parallel_rotations()]
        self.randomize_initial_rotation = randomize_initial_rotation
        self.randomize_initial_position = randomize_initial_position
        self.distance_threshold = distance_threshold
        self.rotation_threshold = rotation_threshold
        self.reward_type = reward_type
        self.ignore_z_target_rotation = ignore_z_target_rotation
        self.target_quat = quat_from_angle_and_axis(0, np.array([.0, .0, 1.]))
        self.initial_quat = quat_from_angle_and_axis(0, np.array([.0, .0, 1.]))
        self.current_level = 0
        self.all_goals = [None] * len(target_positions)
        self.quat_diff = None

        # assert self.target_position in ['ignore', 'fixed', 'random']
        # assert self.target_rotation in ['ignore', 'fixed', 'xyz', 'z',
'parallel']
        initial_qpos = initial_qpos or {}

        hand_env.HandEnv.__init__(
            self, model_path, n_substeps=n_substeps, initial_qpos=initial_qpos,
            relative_control=relative_control)


    def _get_achieved_goal(self):
        # Object position and rotation.
        object_qpos = self.sim.data.get_joint_qpos('object:joint')
        assert object_qpos.shape == (7,)
```

```
        return object_qpos

    def _goal_distance(self, goal_a, goal_b):
        assert goal_a.shape == goal_b.shape
        assert goal_a.shape[-1] == 7

        d_pos = np.zeros_like(goal_a[..., 0])
        d_rot = np.zeros_like(goal_b[..., 0])
        if self.target_positions[self.current_level] != 'ignore':
            delta_pos = goal_a[..., :3] - goal_b[..., :3]
            d_pos = np.linalg.norm(delta_pos, axis=-1)

        if self.target_rotations[self.current_level] != 'ignore':
            quat_a, quat_b = goal_a[..., 3:], goal_b[..., 3:]

            if self.ignore_z_target_rotation:
                euler_a = rotations.quat2euler(quat_a)
                euler_b = rotations.quat2euler(quat_b)
                euler_a[2] = euler_b[2]
                quat_a = rotations.euler2quat(euler_a)

            # Subtract quaternions and extract angle between them.
            quat_diff = rotations.quat_mul(quat_a,
rotations.quat_conjugate(quat_b))
            angle_diff = 2 * np.arccos(np.clip(quat_diff[..., 0], -1., 1.))
            d_rot = angle_diff
        assert d_pos.shape == d_rot.shape
        return d_pos, d_rot

    # GoalEnv methods
    # ----------------------------

    def compute_reward(self, achieved_goal, goal, info = None):
        if self.reward_type == 'sparse':
            success = self._is_success(achieved_goal, goal).astype(np.float32)
            return (success - 1.)
        else:
            d_pos, d_rot = self._goal_distance(achieved_goal, goal)
            # We weigh the difference in position to avoid that `d_pos` (in
meters) is completely
            # dominated by `d_rot` (in radians).
            return -(10. * d_pos + d_rot)

    # RobotEnv methods
    # ----------------------------

    def _is_success(self, achieved_goal, desired_goal):
        d_pos, d_rot = self._goal_distance(achieved_goal, desired_goal)
        achieved_pos = (d_pos < self.distance_threshold).astype(np.float32)
        achieved_rot = (d_rot < self.rotation_threshold).astype(np.float32)
        achieved_both = achieved_pos * achieved_rot
        return achieved_both

    def _env_setup(self, initial_qpos):
        for name, value in initial_qpos.items():
            self.sim.data.set_joint_qpos(name, value)
        self.sim.forward()

    def step(self, action):
```

```
        action = np.clip(action, self.action_space.low, self.action_space.high)
        self._set_action(action)
        self.sim.step()
        self._step_callback()
        obs = self._get_obs()

        done = False
        is_success = self._is_success(obs['achieved_goal'],
self.all_goals[self.current_level])
        reward = self.compute_reward(obs['achieved_goal'],
self.all_goals[self.current_level])
        if is_success:
            if self.current_level < len(self.target_rotations) - 1 :
                self.current_level += 1
            else:
                done = True
        info = {
            'is_success': done,
            'level': self.current_level,
        }
        return obs, reward, False, info

    def _reset_sim(self):
        self.sim.set_state(self.initial_state)
        self.sim.forward()
        self.current_level = 0

        initial_qpos = self.sim.data.get_joint_qpos('object:joint').copy()
        initial_pos, initial_quat = initial_qpos[:3], initial_qpos[3:]
        assert initial_qpos.shape == (7,)
        assert initial_pos.shape == (3,)
        assert initial_quat.shape == (4,)
        initial_qpos = None

        # Randomization initial rotation.
        if self.randomize_initial_rotation:
            if self.target_rotations[0] in ['z', 'z_from_any']:
                angle = self.np_random.uniform(-np.pi, np.pi)
                axis = np.array([0., 0., 1.])
                offset_quat = quat_from_angle_and_axis(angle, axis)
                initial_quat = rotations.quat_mul(initial_quat, offset_quat)
            elif self.target_rotations[0] == 'parallel':
                angle = self.np_random.uniform(-np.pi, np.pi)
                axis = np.array([0., 0., 1.])
                z_quat = quat_from_angle_and_axis(angle, axis)
                parallel_quat =
self.parallel_quats[self.np_random.randint(len(self.parallel_quats))]
                offset_quat = rotations.quat_mul(z_quat, parallel_quat)
                initial_quat = rotations.quat_mul(initial_quat, offset_quat)
            elif self.target_rotations[0] in ['vertical']:
                initial_quat =
self.parallel_quats[self.np_random.randint(len(self.parallel_quats))]
            elif self.target_rotations[0] == 'nearest_xyz':
                angle = self.np_random.uniform(-np.pi, np.pi)
                axis = self.np_random.uniform(-1., 1., size=3)
                self.target_quat = quat_from_angle_and_axis(angle, axis)
                initial_quat = self.nearestParallel(self.target_quat)
            elif self.target_rotations[0] in ['xyz', 'ignore', 'fixed']:
                angle = self.np_random.uniform(-np.pi, np.pi)
```

```
                axis = self.np_random.uniform(-1., 1., size=3)
                offset_quat = quat_from_angle_and_axis(angle, axis)
                initial_quat = rotations.quat_mul(initial_quat, offset_quat)
            else:
                raise error.Error('Unknown target_rotation option
"{}".'.format(self.target_rotations[0]))

        # Randomize initial position.
        if self.randomize_initial_position:
            # if self.target_position != 'fixed':
            initial_pos += self.np_random.normal(size=3, scale=0.005)

        initial_quat /= np.linalg.norm(initial_quat)
        initial_qpos = np.concatenate([initial_pos, initial_quat])
        self.sim.data.set_joint_qpos('object:joint', initial_qpos)

        def is_on_palm():
            self.sim.forward()
            cube_middle_idx = self.sim.model.site_name2id('object:center')
            cube_middle_pos = self.sim.data.site_xpos[cube_middle_idx]
            is_on_palm = (cube_middle_pos[2] > 0.04)
            return is_on_palm

        # Run the simulation for a bunch of timesteps to let everything settle in.
        for _ in range(10):
            self._set_action(np.zeros(20))
            try:
                self.sim.step()
            except mujoco_py.MujocoException:
                return False
        return is_on_palm()

    def sample_goal_for_level(self, level, parent_goal, initial_pos):
        # Select a goal for the object position.
        target_position = self.target_positions[level]
        target_rotation = self.target_rotations[level]
        target_pos = None
        if target_position == 'random':
            assert self.target_position_range.shape == (3, 2)
            offset = self.np_random.uniform(self.target_position_range[:, 0],
self.target_position_range[:, 1])
            assert offset.shape == (3,)
            target_pos = self.sim.data.get_joint_qpos('object:joint')[:3] + offset
        elif target_position in ['ignore', 'fixed']:
            target_pos = self.sim.data.get_joint_qpos('object:joint')[:3]
        else:
            raise error.Error('Unknown target_position option
"{}".'.format(target_position))
        assert target_pos is not None
        assert target_pos.shape == (3,)

        # Select a goal for the object rotation.
        target_quat = None
        if target_rotation == 'z':
            angle = self.np_random.uniform(-np.pi, np.pi)
            axis = np.array([0., 0., 1.])
            target_quat = quat_from_angle_and_axis(angle, axis)
        elif target_rotation == 'z_from_any':
            if parent_goal is None:
```

```
                angle = self.np_random.choice([0, np.pi / 2, np.pi, -np.pi / 2])
                axis = np.array([.0, .0, 1.])
                target_quat = quat_from_angle_and_axis(angle, axis)
                initial_parallel =
self.nearestParallel(self.sim.data.get_joint_qpos('object:joint')[3:7])
                target_quat = rotations.quat_mul(target_quat, initial_parallel)
            else:
                target_quat = parent_goal[3:7]
        elif target_rotation == 'vertical':
            # if parent_goal is None:
            angle = self.np_random.choice([0, np.pi / 2, np.pi, -np.pi / 2])
            axis = np.array([.1, .0, 0.])
            target_quat = quat_from_angle_and_axis(angle, axis)
            initial_parallel =
self.nearestParallel(self.sim.data.get_joint_qpos('object:joint')[3:7])
            parent_parallel = self.nearestParallel(parent_goal[3:7])
            self.quat_diff =
rotations.quat2euler(rotations.quat_mul(parent_parallel,
rotations.quat_conjugate(initial_parallel)))
            target_quat = rotations.quat_mul(target_quat, initial_parallel)

        elif target_rotation == 'nearest_xyz':
            angle = self.np_random.uniform(-np.pi, np.pi)
            axis = self.np_random.uniform(-1., 1., size=3)
            target_quat = quat_from_angle_and_axis(angle, axis)
        elif target_rotation == 'parallel':
            angle = self.np_random.uniform(-np.pi, np.pi)
            axis = np.array([0., 0., 1.])
            target_quat = quat_from_angle_and_axis(angle, axis)
            parallel_quat =
self.parallel_quats[self.np_random.randint(len(self.parallel_quats))]
            target_quat = rotations.quat_mul(target_quat, parallel_quat)
        elif target_rotation == 'xyz':
            angle = self.np_random.uniform(-np.pi, np.pi)
            axis = self.np_random.uniform(-1., 1., size=3)
            target_quat = quat_from_angle_and_axis(angle, axis)
        elif target_rotation in ['ignore', 'fixed']:
            target_quat = self.sim.data.get_joint_qpos('object:joint')[3:7]
        else:
            raise error.Error('Unknown target_rotation option
"{}".'.format(target_rotation))
        assert target_quat is not None
        assert target_quat.shape == (4,)

        target_quat /= np.linalg.norm(target_quat)  # normalized quaternion
        goal = np.concatenate([target_pos, target_quat])
        return goal

    def _sample_goal(self):
        target_pos = self.sim.data.get_joint_qpos('object:joint')[:3]
        initial_quat =
self.nearestParallel(self.sim.data.get_joint_qpos('object:joint')[3:7])
        angle = self.np_random.choice([0, np.pi / 2, np.pi, -np.pi / 2])
        axis = np.array([0., 0., 1.])
        middle_quat = quat_from_angle_and_axis(angle, axis)
        target_quat = rotations.quat_mul(middle_quat, initial_quat)
        self.all_goals[0] = np.concatenate([target_pos, target_quat])
        angle = np.pi  #self.np_random.choice([0, np.pi / 2, np.pi, -np.pi / 2])
        axis = np.array([1., 0., 0.])
```

```
        middle_quat = quat_from_angle_and_axis(angle, axis)
        target_quat = rotations.quat_mul(middle_quat, target_quat)
        self.all_goals[1] = np.concatenate([target_pos, target_quat])
        angle = self.np_random.uniform(-0.6, 0.6)
        axis = self.np_random.uniform(-1., 1., size=3)
        middle_quat = quat_from_angle_and_axis(angle, axis)
        target_quat = rotations.quat_mul(middle_quat, target_quat)
        self.all_goals[2] = np.concatenate([target_pos, target_quat])
        return self.all_goals[0]

    def nearestParallel(self, initial_pos):
        target_rotation_parallel = None
        min_angle = 999
        for rotation in self.parallel_quats:
            quat_diff = rotations.quat_mul(rotation,
rotations.quat_conjugate(initial_pos))
            angle_diff = 2 * np.arccos(np.clip(quat_diff[..., 0], -1., 1.))
            if angle_diff < min_angle:
                target_rotation_parallel = rotation
                min_angle = angle_diff

        axis = np.array([0., 0., 1.])
        target_quat = quat_from_angle_and_axis(0, axis)
        return rotations.quat_mul(target_quat, target_rotation_parallel)

   def _render_callback(self):
        # Assign current state to target object but offset a bit so that the
actual object
        # is not obscured.
        goal = self.all_goals[self.current_level].copy()
        mid_goal = self.all_goals[2].copy()

        assert goal.shape == (7,)
        if self.target_positions[self.current_level] == 'ignore':
            # Move the object to the side since we do not care about it's
position.
            goal[0] += 0.15
            mid_goal[0] += 0.25
        self.sim.data.set_joint_qpos('target:joint', goal)
        self.sim.data.set_joint_qvel('target:joint', np.zeros(6))
        self.sim.data.set_joint_qpos('mid_target:joint', mid_goal)
        self.sim.data.set_joint_qvel('mid_target:joint', np.zeros(6))

        if 'object_hidden' in self.sim.model.geom_names:
            hidden_id = self.sim.model.geom_name2id('object_hidden')
            self.sim.model.geom_rgba[hidden_id, 3] = 1.
        self.sim.forward()

    def _get_obs(self):
        robot_qpos, robot_qvel = robot_get_obs(self.sim)
        object_qvel = self.sim.data.get_joint_qvel('object:joint')
        achieved_goal = self._get_achieved_goal().ravel()  # this contains the
object position + rotation
        observation = np.concatenate([robot_qpos, robot_qvel, object_qvel,
achieved_goal])
        return {
            'observation': observation.copy(),
            'achieved_goal': achieved_goal.copy(),
            'desired_goal': self.all_goals[self.current_level].ravel().copy()
```

```
        }
class HandBlockEnvHierarchy(ManipulateEnv, utils.EzPickle):
    def __init__(self, target_positions=['random'], target_rotations=['xyz'],
reward_type='sparse'):
        utils.EzPickle.__init__(self, target_positions, target_rotations,
reward_type)
        ManipulateEnv.__init__(self,
            model_path=MANIPULATE_BLOCK_XML, target_positions=target_positions,
            target_rotations=target_rotations,
            target_position_range=np.array([(-0.04, 0.04), (-0.06, 0.02), (0.0,
0.06)]),
            reward_type=reward_type)
```